NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

THE UNEXPLORED IMPACT OF IPV6 ON INTRUSION DETECTION SYSTEMS

by

Keith A. Gehrke

March 2012

Thesis Advisor:	Robert Beverly
Second Reader:	J.D. Fulp

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
26–3–2012	Master's Thesis	2010-01-04—2012-03-30

4. TITLE AND SUBTITLE

The Unexplored Impact of IPv6 on Intrusion Detection Systems

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Keith A. Gehrke

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Postgraduate School
Monterey, CA 93943

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Department of the Navy

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited

13. SUPPLEMENTARY NOTES

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. **IRB Protocol Number: XXXX**

14. ABSTRACT

With DoD networks steadily adopting and transitioning to the next generation Internet Protocol, IPv6, careful consideration must be given to IPv6-specific implications on network protection. While Network Intrusion Detection Systems (NIDS) assist in protecting current IPv4 DoD networks, NIDS performance in operational DoD IPv6 environments is largely unknown. As a step toward more rigorous NIDS evaluation, we investigate the extent to which known IPv4 attacks are able to evade detection when converted to equivalent IPv6 attacks. Utilizing 13 general attack classes, we test the IPv6 readiness of two popular open source NIDSs: SNORT and BRO. Attacks in each class are evaluated in a virtual test bed that models both "native" and "transitional" networks. In the native IPv6 environment, we achieve a 95% detection rate for SNORT as compared to 8% with BRO. In addition, we discover a bug in SNORT where a carefully crafted IPv6 packet causes the NIDS to fail open, allowing full circumvention. Our findings suggest that, with respect to IPv6, both NIDS signatures and NIDS software require additional testing and evaluation to be operationally ready.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			19b. TELEPHONE NUMBER *(include area code)*
Unclassified	Unclassified	Unclassified	UU	117	

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

THE UNEXPLORED IMPACT OF IPV6 ON INTRUSION DETECTION SYSTEMS

Keith A. Gehrke
Lieutenant, United States Navy
B.S., University of Phoenix, 2002

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 2012

Author: Keith A. Gehrke

Approved by: Robert Beverly
Thesis Advisor

J.D. Fulp
Second Reader

Peter J. Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

With DoD networks steadily adopting and transitioning to the next generation Internet Protocol, IPv6, careful consideration must be given to IPv6-specific implications on network protection. While Network Intrusion Detection Systems (NIDS) assist in protecting current IPv4 DoD networks, NIDS performance in operational DoD IPv6 environments is largely unknown. As a step toward more rigorous NIDS evaluation, we investigate the extent to which known IPv4 attacks are able to evade detection when converted to equivalent IPv6 attacks. Utilizing 13 general attack classes, we test the IPv6 readiness of two popular open source NIDSs: SNORT and BRO. Attacks in each class are evaluated in a virtual test bed that models both "native" and "transitional" networks. In the native IPv6 environment, we achieve a 95% detection rate for SNORT as compared to 8% with BRO. In addition, we discover a bug in SNORT where a carefully crafted IPv6 packet causes the NIDS to fail open, allowing full circumvention. Our findings suggest that, with respect to IPv6, both NIDS signatures and NIDS software require additional testing and evaluation to be operationally ready.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms

AH Authentication Header

ARP Address Resolution Protocol

ASCII American Standard Code for Information Interchange

CA Certification Authority

CGA Cryptographically Generated Addresses

DAD Duplicate Address Detection

DHCP Dynamic Host Configuration Protocol

DNS Domain Name Service

DoD Department of Defense

DoS Denial of Service

DDoS Distributed Denial of Service

ESP Encapsulating Security Protocol

FTP File Transfer Protocol

GDB GNU Project Debugger

ICMP Internet Control Message Protocol

IANA Internet Assigned Numbers Authority

IDS Intrusion Detection System

IP Internet Protocol

IPS Intrusion Prevention System

IPsec Internet Protocol Security

IPv4 Internet Protocol Version 4

IPv4 Internet Protocol Version 6

IRC Internet Relay Chat

LAN Local Area Network

MiTM Man in the Middle

MLD Multicast Listener Discovery

MTU Maximum Transmission Unit

NA Neighbor Advertisement

NAT Network Address Translation

ND Neighbor Discovery

NDP Neighbor Discovery Protocol

NFA Non-Deterministic Finite Automata

NIDS Network Intrusion Detection System

NIQ Node Information Query

NIST National Institute of Standards and Technology

NS Neighbor Solicitation

OS Operating System

QoS Quality of Service

PCRE Perl Compatible Regular Expressions

PMTUD Path MTU Discovery

RA Router Advertisements

RFC Request For Comment

RIR Regional Internet Registry

RS Router Solicitation

SEND Secure Neighbor Discovery

SLAAC Stateless Automatic Address Configuration

TCP Transport Control Protocol

THC The Hackers Choice

UDP User Datagram Protocol

Acknowledgements

This thesis was made possible by Dr. Rob Beverly and Mr. J.D. Fulp. The guidance, direction, and attention to detail that they have contributed is very appreciated. To both of you, I give a sincere thank you.

To my wonderful family, especially my beautiful wife Katrina, I owe you all a tremendous amount of gratitude. The amount of sacrifice required by you during this process was more than I could have asked for, yet you gave it unquestionably. Your support and caring only proves how lucky I am to have you all, thank you.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1:
Introduction

It is inevitable. Internet Protocol version 6 (IPv6) [1] will replace Internet Protocol version 4 (IPv4) [2] as the protocol backbone of our packet-switched networks, devices, and eventually the Internet. In fact, with the last IPv4 /8 address blocks being issued in early February 2011 and the 8 June 2011 world IPv6 day (discussed in §2.3), the start-again stop-again conversion to IPv6 is rapidly approaching [3]. This movement towards IPv6 brings to light the need for knowledge of IPv6 infrastructure and how to secure it (e.g., Network Intrusion Detection, Firewalls, Access Control Lists). To date, these remain mostly little explored domains. Industry (e.g., Google, Akamai, Yahoo, Ericsson, and Cisco) has made the move towards IPv6 with events like World IPv6 day, and now it is time for everyone else to get on board. This includes the federal government and Department of Defense infrastructure, which must be prepared for IPv6 in order to prevent having to catch up on the trend.

IPv4 address exhaustion has been and continues to be an increasingly serious problem. Recently the Regional Internet Registries (RIRs) have begun to allow IPv4 address holders to sell address blocks in order to free up claimed (but un-used) IPv4 address space, thus creating a market like scenario. Stop-gap measures tried in the past along with those tried today, such as allowing an IPv4 address market, only buys time and does not solve the fundamental address scarcity [4]. In fact, as IPv4 address scarcity increases, the mechanisms designed to cope with IPv4 address exhaustion (e.g., NAT) are no longer good enough. These coping mechanisms limit end-to-end reachability, application functionally, and stifle innovation (particularly in the content industry). With the economic push from industry stake holders such as content providers, the driving force behind IPv6 adoption today is fundamentally different from that of even 5 years ago.

An August 2005 memorandum to all U.S. Federal Government Chief Information Officers required systems to transition to IPv6 by June 2008 [5]. Despite sporadic adoption, this mandate has yet to take place. The on-again and off-again nature of the pending need for IPv6 has made many in the military networking community slow to focus any attention on IPv6. However, this change is coming. All DoD networks will, in the future, be operating in an IPv6 environment and must be protected from adversaries who are rapidly gaining knowledge of IPv6 weaknesses and exploits (§2.3). Amid the current conversion to IPv6, few have considered the associated vulnerabilities. A focused effort must be placed on how IPv6 changes the DoD network operat-

ing environment. We therefore examine how to protect DoD networks and information, before, during, and after the IPv6 transition process.

Current IPv6 security efforts have focused primarily on attack theory and vulnerabilities. Specifically, the impact of IPv6 attacks on host computers, the network stack, and operating systems (OS). This thesis will explore the impact of IPv6 attacks on Network Intrusion Detection Systems (NIDS) as well as the the possibility of new IPv6 attacks.

1.1 Background of IPv6 and NIDS Readiness Posture

This thesis will focus on the exploration of the impact IPv6 vulnerabilities and attacks will have on NIDSs. Even though many NIDSs support IPv6, there are common misconceptions about the level of security provided by the existing NIDS signatures available to detect attacks in an IPv6 environment. While determining the effect a vulnerability or attack will have on a particular host can show the impact on a specific operating system (OS), router or computer, doing so does not aid in the detection or prevention of these vulnerabilities or attacks. A determination of the NIDSs "IPv6 readiness" must be conducted. To evaluate readiness, each NIDS must be tested against IPv6 vulnerabilities and attacks, both theoretical and actual.

Another common misconception is that IPv6 is secure and there is no need to be concerned with its vulnerabilities or possible attacks. This is not the case, for example any application layer attack possible in IPv4 will still be just as dangerous in IPv6 [6]. Flooding (§2.4.1) type attacks like Denial of Service (DoS) attacks will still be a threat to our networks, just as they were in IPv4 [7]. Networks will still be susceptible to Man-in-the-Middle attacks (§2.4.1), unauthorized access (§2.4.2), as well as attacks against the Physical or Data-link layers, as we was the case with IPv4 [8]. Any expectation that IPSec [9] combined with the increased size of the IPv6 address space will cure existing network security problems when we "go native" IPv6, is an oversight. The misconception that the large IPv6 address space hides hosts from malicious scanners, without any further detection or prevention, is dangerous [5]. These short-sited beliefs are in themselves vulnerabilities that can be expected to be exploited.

The adoption of IPv6 presents new and unique challenges to which the industry is just beginning to adapt. To accurately assess the risks on networks and the impact on the NIDS, we must first look at existing threats that have been changed due to protocol differences between IPv6 and IPv4. Some existing IPv4 threats may map directly to IPv6, some may require modification, and entirely new IPv6-specific threats may be introduced. These threats include,

but are not necessarily limited to, LAN-based attacks (Address Resolution Protocol [ARP] or Neighbor Discovery Protocol [NDP] §2.1.4), attacks against DHCP (for IPv6 DHCPv6, DoS attacks against routers (hop-by-hop extension headers rather than router alerts), and fragmentation §2.4.2 (IPv4 routers performing fragmentation versus IPv6 hosts via extension headers)) [8]. Next, the threats unique to IPv6 networks must be examined. Attacks focused against ICMPv6 (§2.1.3), auto configuration (§2.1.1), IPv6 multicast (§2.1.1), and extension headers (§2.1.2) present targets of opportunity for the hackers interested in IPv6 because its traffic is not tracked on par with IPv4, and the security measures in place for IPv4 are not present for IPv6; making it an ideal covert communications channel [5]. Finally, IPv6 fuzz testing [10] should be performed on the IPv6 NIDS implementations, specifically targeting the impact on the NIDS, in an effort to determine the effect of IPv6-unique vulnerabilities.

1.2 Research Questions

This thesis asks two primary questions: which known IPv4 exploits are feasible or infeasible when converted to IPv6? And will popular open source NIDSs detect these IPv6 attacks? These primary objectives can be achieved through answering the secondary questions:

- What are the current IPv6 exploits?
- What exploits are newly enabled by IPv6?
- What is the impact of automated IPv6 fuzz testing on the NIDSs?

These questions are answered in a controlled virtual IPv6 test bed that permits repeatable testing in a sanitized environment. The test bed transmits converted attacks between hosts. This thesis considers two popular open-source Network Intrusion Detection Systems: SNORT and BRO. The NIDS' passively observe traffic on the subnet between hosts. These experiments provide detailed results of IPv6 attack to NIDS detection, or NIDS detection profiles, that can be used in future research for the development of signatures and, where feasible, patches for given attacks.

1.3 Significant Findings

On the basis of an attack matrix (§3.3) developed from thirteen general attack classes, two open source NIDSs were used in a controlled test bed, configured, and tested in two IPv6 environments. More than 80 controlled tests were conducted, 40 "Native" and 40 "Transitional",

producing two detection matrices. Fuzz testing of NIDS reactions to changes in packets was also conducted. Careful analysis of all data revealed:

- SNORT 2.9.0.5 in the "Native" environment produced a 95 percent detection success rate, and a 93 percent detection success rate in the Transitional environment.
- In the "Native" environment both BRO 1.5.3 and BRO version 2.0 produced an eight percent detection success rate, while in the Transitional environment they produced a five percent detection success rate.
- Discovered a vulnerability in SNORT 2.9.0.5 when sending a large number of hop-by-hop extension headers. This bug, located in *Decode.c*, allows for an attacker to put SNORT into a state of infinite recursion that prevents it from processing any subsequent packets, thereby allowing an attacker to circumvent the NIDS.
- A Bug in *net_util.cc* produces a "bad dotted address" error message when BRO 1.5.3 is processing IPv6 FTP traffic.
- For both versions of BRO, not all IPv6 events are processed. Due to a problem with IPv6 data flow, events are not making it to the Event Handler which in most cases prevented detection.

1.4 Thesis Structure

The remainder of this thesis is organized as follows:

- Chapter 2 covers the basics of IPv6, differences between IPv4 and IPv6, Network Intrusion Detection Systems, defines the List of Exploits to be used, as well as gives a brief description of Fuzz testing.
- Chapter 3 discusses the IPv6 testbed configuration for the experiments and discusses the exploits used against the NIDSs in the experiments.
- Chapter 4 provides the results of experiments and contains detection profiles for each NIDS.
- Chapter 5 contains conclusions drawn from Chapter 4 and recommended future work.

CHAPTER 2:
Background and Related Work

This chapter is intended to provide an overview of IPv6 while highlighting some of the protocol differences from the current main stream Internet protocol, IPv4 [2]. This chapter also provides overviews of Network Intrusion Detection Systems (NIDS), relevant IPv4 and IPv6 exploits, automated protocol fuzz testing, and the IPv6 capabilities of todays NIDSs. The list of attacks described in this section will be further defined in later chapters and is meant to set a base line for the reader to comprehend the possibilities and impact of each attack.

2.1 IPv6 Overview

IPv6 is the new version of the Internet Protocol, designed as the successor to IP version 4 (IPv4) [2]. The changes from IPv4 to IPv6 fall primarily into the following categories: [1]

- Expanded Addressing Capabilities
- Header Format Simplification
- Improved Support for Extensions and Options
- Flow Labeling Capability
- Authentication and Privacy Capabilities

In comparison to IPv4, IPv6 provides many improvements with respect to simplicity, routing speed, quality of service (QoS), and security [1]. IPv6 has the potential to improve security and confidentiality of transmitted information by utilizing its built-in security mechanisms. Both IPv4 and IPv6 are layer three (network layer) routing protocols that rely on an addressing scheme. However, IPv6 was designed to resist the need for additional new features and to have minimal impact on upper and lower level protocols. The IPv6 protocol has a new header format, larger address space, efficient hierarchical addressing and routing infrastructure, stateless and stateful address configuration, built-in security (IPsec), better support for QoS, new protocols for neighboring node interaction, and extensibility [11]. IPv6 was designed as an upgrade or next iteration of IPv4. In fact careful consideration was given to the fact that transition from

IPv4 to IPv6 may be a slow process drawn out over time, which meant that transition mechanisms had to be built in, and IPv4 and IPv6 needed to be able to work in tandem. Even though IPv6 has many upgrades from its predecessor, it shares IPv4's foundation, adopting some of its characteristics, strengths, and weakness. This rings true from the average user's perspective, where the transition to IPv6 from IPv4 will be completely transparent, leaving them with no idea of which protocol they are using nor what its capabilities, limitations or vulnerabilities are. Thus, while IPv6 has fixed some of IPv4's problems, for example the relatively small address space, it introduces its own issues and problems to be considered.

2.1.1 Addressing

Address Space

Unlike IPv4 with its 32 bit addressing space, IPv6 uses 128 bits to define its address space. This produces vast numbers of addresses that dwarf those available in IPv4. For example if every subscriber was given a /48 Global Unicast Address prefix, each would contain 45 variable bits. This means that each subscriber would have 48 bits minus the 3 bit type field, which leaves each subscriber with 45 variable bits from which addresses can be derived, see Figure 2.2. That is, the number of available prefixes is 2^{45} or about 35 trillion [12] .This increase from 32 bits to 128 bits also provides more levels of addressing hierarchy, a greater number of addressable nodes, and a simplified auto-configuration of addresses; all of which provide the network engineer increased flexibility. Specifically, the larger address space allows for many more devices and users on the internet as well as extra flexibility in allocating addresses, thus eliminating the need for address conservation practices (e.g., NAT) and simplifying the auto-configuration process. IPv6 addresses identify interfaces within one of three hierarchical regions of the network. The scope of an address could be link-local, site-local, or global [13].

IP Addresses

As with IPv4, IPv6 addresses have the most-significant part of the address placed to the left, allowing for easy recognition of various address formats when logically dividing the 128 address bits into bit groups that can then be associated with special addressing features. The largest group of IPv6 addresses are global unicast addresses. The rest of the address groups are composed of unspecified and loopback addresses, multicast addresses, and link- and site-local addresses. Some of the ranges of addresses currently in use are illustrated in Figure 2.1.

::	Unspecified address
::1	Loopback address
2001::/16	"Sub-TLA" (RFC 2450) normal addresses
2002::/16	"6to4" (RFC 3056) automatic tunnels
3FFE::/16	"6bone" (RFC 2471) testing addresses
FE80::/10	Link-local addresses
FEC0::/10	Site-local addresses
FF00::/8	Multicast addresses

Figure 2.1: IPv6 address space. From [8]

As shown in Figure 2.1 there are three types of IPv6 addresses: [14]

- **Unicast:** An identifier for a single interface. A packet sent to a unicast address is delivered to the interface identified by that address.
- **Anycast:** An identifier for a set of interfaces (typically belonging to different nodes). A packet sent to an anycast address is delivered to any one of the destination nodes identified by that address (the ”nearest” one, according to the routing protocols’ measure of the distance).
- **Multicast:** An identifier for a set of interfaces (typically belonging to different nodes). A packet sent to a multicast address is delivered to all interfaces identified by that address.

In IPv6 broadcast addresses are not available; instead multicast is used. For example, the ”link-scope all-hosts multicast” address, `ff02::1`, corresponds to the IPv4 subnet-local broadcast address, `255.255.255.255`.

All IPv6 interfaces are required to have at least one link-local unicast address. A single interface may also have multiple IPv6 addresses of any type (Unicast, Multicast, Anycast) or scope [14]. Additionally multicast routing scalability is improved in IPv6 due to the addition of an added “scope” field to each multicast address. The addition of the “anycast address” is defined to allow for the delivery of a packet to any one of a group of nodes [1]. The IPv6 address is broken up into two 64 bit sections, the first is for network identification, while the second half identifies the host or Interface. Figure 2.2 shows the IPv6 Global Unicast Address breakdown, while Figure 2.3 shows the breakdown of the IPv6 Link Local Address.

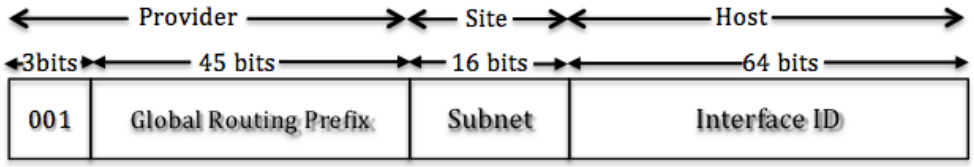

Figure 2.2: Example of global unicast IPv6 address.

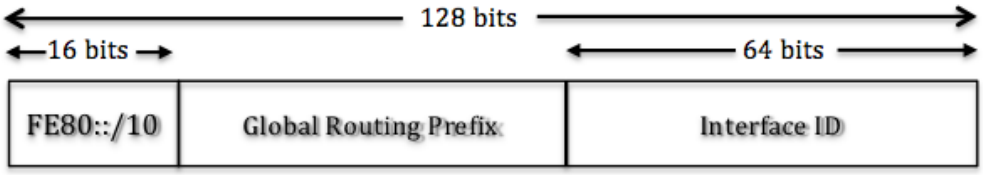

Figure 2.3: Example of link local IPv6 address.

Host Auto-configuration

An IPv6 host interface can have multiple unicast addresses, a link-local address (which is the first address assigned to the interface), and one or many global or site-local addresses. Configuration of interfaces in IPv6 is controlled by the protocol itself [13]. The host auto-configuration feature allows hosts joining a link to configure link-local addresses for their interfaces as well as to check the uniqueness and validity of assigned addresses. Stateless auto-configuration is the process that allows an IPv6 host to be assigned addresses based on local router advertisements (RA) §2.1.4. In contrast, IPv4 uses the stateful address auto-configuration protocol, or Dynamic Host Configuration Protocol (DHCP). In the stateful auto-configuration model, a host obtains the interface addresses as well as other required information such as the address of the default gateway a DNS server from a DHCP server. The DHCP server maintains a manually administered list of hosts and keeps track of which addresses have been assigned to which hosts. In addition, IPv6 offers stateless DHCPv6 which is a procedure during which addresses are configured according to the router advertisements along with additional information given to the host, such as default gateway and DNS servers, via a DHCP server.

2.1.2 IPv6 Header

The IPv6 protocol header is the first place where noticeable differences from IPv4 exist. One of the key differences is in the Source and Destination addresses. Whereas IPv4 uses only 32

bit addresses, IPv6 uses 128 bit addresses. In addition, some of the IPv4 header fields have been dropped or made optional. Examples of these dropped fields include, Flags, Identifier, and Checksum. While the Fragment and Options and Padding fields have been replaced by IPv6 extension headers. This is to reduce the common-case processing cost of packet handling and to limit the bandwidth cost of the IPv6 header [1]. The specific fields of the IPv6 protocol header are shown in Figure 2.4. [8]

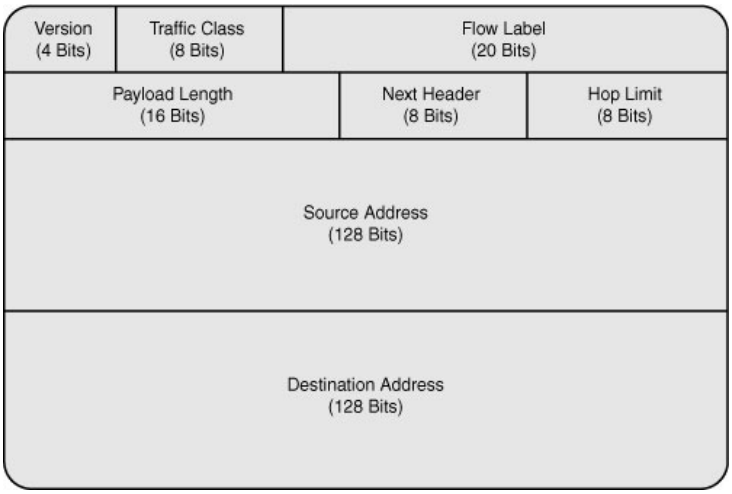

Figure 2.4: IPv6 protocol header. From [8]

The fields within the IPv6 header each have very specific jobs. Here is a list of those fields and their use:

- **Version:** Always equal to 6 for IPv6.
- **Traffic Class:** Identifies the priority and class of service of this packet.
- **Flow Label:** For future use in identifying packets that are part of a unique flow, stream, or connection.
- **Payload Length:** Defines the length in octets of the packet that follows the IPv6 header.
- **Next Header:** Identifies the type of header that follows the IPv6 header.
- **Hop Limit:** Counter for the remaining number of hops that the packet can traverse.
- **Source Address:** The IPv6 address of the node that originated this packet.
- **Destination Address:** The IPv6 address that this packet is destined for.

When comparing the IPv4 header to the IPv6 header it is noticeable that the IPv6 header is cleaner, with fewer fields and is aligned to support the current processors found in most host computers and devices by using 4 and 8 bit boundaries that make header disassembly more efficient. The IPv6 header has also been made more simple by removing all the fields (from IPv4) that have little to no use in the v6 version of the protocol. In an attempt improve performance the IPv6 header only contains essential data with everything else (e.g., fragmentation) handled by extension headers. Overall, the IPv6 header has been streamlined for simplicity and performance.

Extension Headers

In IPv6, extension headers are used to indicate the transport layer information of the packet (TCP or UDP), or to extend the functionality of the protocol. Extension headers are identified with the Next Header (NH) field within the IPv6 header. This field is similar to the Protocol field in an IPv4 packet [8]. The next header field, which is 8 bits long, identifies the header following the IPv6 header. These optional headers indicate what type of information follows the IPv6 header.

Extension headers follow the IPv6 header and are a sequential list of optional internet layer information that are encoded in separate headers that may be placed between the IPv6 header and the transport layer header in the packet [1]. The typical format of an extension header is an 8-bit option type that contains the number of the next extension header in the list, an 8-bit unsigned integer of option data length that tells how long the header is, and the option data payload that is of variable size. Extension headers can be combined, where several appear concatenated (or "chained") in a single packet, but typically only a few are included [8]. The structure and arrangement of extension headers is illustrated in Figure 2.5.

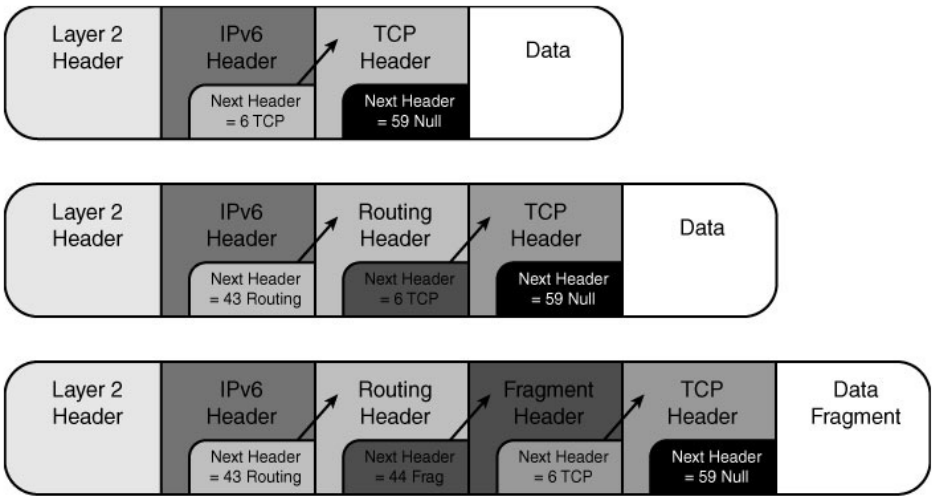

Figure 2.5: Extension headers. From [8]

An IPv6 packet may carry zero or more extension headers, each identified by the Next Header field of the preceding header. With one exception, the Hop-by-Hop option header, extension headers are not examined or processed by any node along a packet's delivery path, until the packet reaches the node (or each of the set of nodes, in the case of multicast) identified in the Destination Address field of the IPv6 header [1]. The contents of each extension header determines where to proceed to the next header, or if the Next Header value is Null continue to process the packets data. The use of extension headers follow strict rules and order of precedence. The following rules apply to the use of extension headers:

- Each extension header should not appear more than once, with the exception of the Destination Options header.
- The Hop-by-Hop Options header should only appear once.
- The Hop-by-Hop Options header should be the first header in the list because it is examined by every node along the path.
- The Destination Options header should appear a most twice (before a Routing header and before the upper-layer header).
- The Destination Options header should be the last header in the list, if it is used at all.
- The Fragment header should not appear more that once and should not be combined with the Jumbo Payload Hop-by-Hop option.

Because extension headers have a specific order, as defined in RFC 2460, they must be processed in the order they appear in the packet. This order of precedence is defined in Table 2.1.

1	IPv6 Header
2	Hop-by-Hop Options header
3	Destination Options header
4	Routing header
5	Fragment header
6	Authentication header
7	Encapsulation Security Payload header
8	Destination Options header
9	Upper-layer header

Table 2.1: Extension header precedence

Since the extension header must be parsed with the Next Header field to determine what to do next each extension header has a unique number used in the preceding header's Next Header field. This allows the receiving node to know how to parse the header to follow. These numbers are defined by IANA [15] and follow the IPv4 protocol numbers. For more information on Extension headers or protocol numbers see RFC2460 [1] or [15].

Fragment Header

"Fragmentation" is the term given to the process of breaking down an IP datagram into smaller packets, each with its own packet header, to be transmitted over different types of network media and then reassembling them at the other end. This process is an integral part of the IP protocol and is covered in depth in [2]. Or restated, it is the process of dissecting the packet into smaller packets to be easily carried across a data network that does not have the capability to carry large packets. Fragmentation occurs due to networks with varying sizes of Maximum Transmission Units (MTUs) [8].

The Fragment Header is used by an IPv6 source to send a packet larger than would fit in the path MTU to its destination [1]. Unlike IPv4, fragmentation in IPv6 is performed only by source node and not by routers along the path (in-network fragmentation). Each packet receives a unique Fragment Identifier and is identified by the value 44 in the preceding Next Header field. In order to send any packet that is too large, the source node may divide the packet into fragments and send each fragment as a separate packet, which is then reassembled at the destination node. Figure 2.6 illustrates how the large packet (top) needs to be fragmented into the two smaller packets (bottom). The original packet is made up of an unfragmentable part that

contains the original IPv6 header plus any extension headers that must be processed by nodes en route to the destination. The fragmentable part is fragmented creating multiple packets, each having the unfragmentable part, including any headers that are required (e.g., routing header or Hop by Hop extension header), as well as a fragment header. [8]

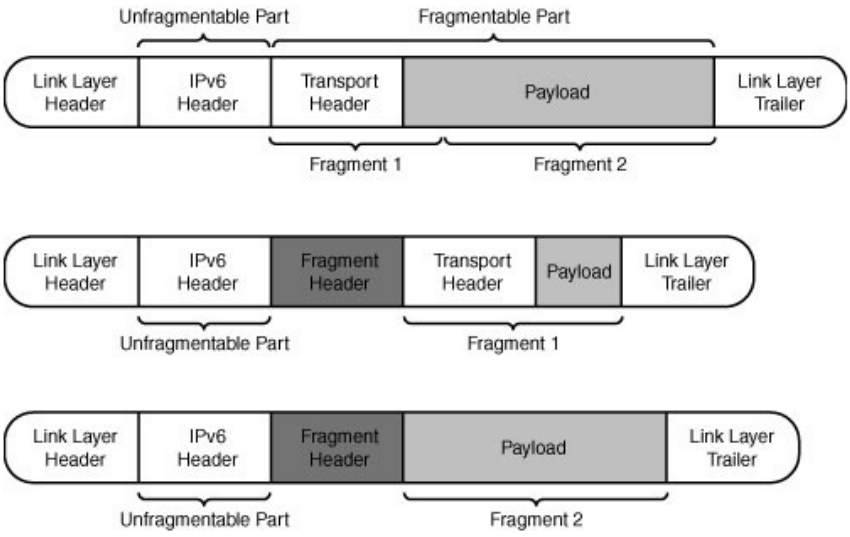

Figure 2.6: Fragmented packet. From [8]

2.1.3 ICMPv6

IPv6 uses the Internet Control Message Protocol (ICMP6) as defined for IPv4 [16], with a number of changes [17]. ICMPv6 is vital to the proper operation of the IPv6 protocol. Unlike ICMP for IPv4, which is not required for IPv4 communications, ICMPv6 has features that are required elements which cannot be completely filtered [8]. For example the host auto-configuration and Neighbor Discovery Protocol (NDP) §2.1.4 both require ICMPv6 messages to be able to complete address assignments and perform Duplicate Address Detection (DAD); both of which are vital to IPv6 operation. ICMPv6 operates on top of IPv6 as an extension header but actually works in conjunction with IPv6 for protocol operations. ICMPv6 is an integral part of IPv6, and must be fully implemented by every IPv6 node [17].

ICMPv6 is used by IPv6 nodes to report errors encountered in processing packets, and to perform other internet-layer functions, such as diagnostics and testing (e.g., `traceroute6`). ICMPv6 messages contain a type and a code that relate the details of the message to the type of message, as well as a checksum and a payload of variable size. ICMPv6 error messages re-

lay useful information back to the source of the packet about any error that may have occurred along the path. The general packet structure for ICMPv6 is shown in Figure 2.7.

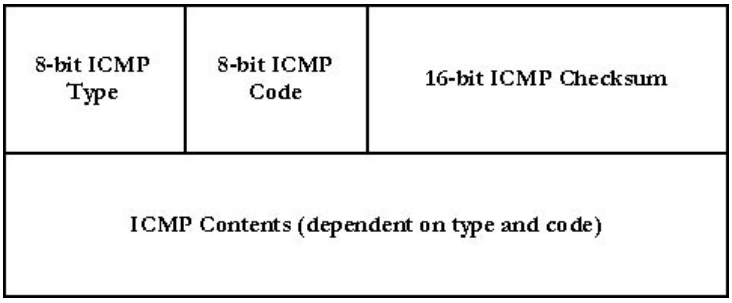

Figure 2.7: General ICMPv6 packet.

- **Type:** The type field indicates the type of message.
- **Code:** The code field depends on the message type and is used to create an additional level of message granularity.
- **Checksum:** The checksum field is used to detect data corruption in the ICMPv6 message and parts of the IPv6 header. (for more information on how to calculate the checksum see RFC 4443)

ICMPv6 can be considered as the backbone of the IPv6 protocol, providing the following functions: [8]

- Neighbor Discovery Protocol (NDP), Neighbor Advertisements (NA), and Neighbor Solicitations (NS) provide the IPv6 equivalent of IPv4 Address Resolution Protocol (ARP) functionality.
- Router Advertisements (RA) and Router Solicitations (RS) help nodes determine information about their LAN, such as the network prefix, the default gateway, and other information that can help them communicate.
- Echo Request and Echo Reply support the Ping6 utility.
- PMTUD determines the proper MTU size for communications.
- Multicast Listener Discovery (MLD) provides IGMP-like functionality for communicating IP multicast joins and leaves.

- Multicast Router Discovery (MRD) discovers multicast routers.
- Node Information Query (NIQ) shares information about nodes between nodes.
- Secure Neighbor Discovery (SEND) helps secure communications between neighbors.

2.1.4 Neighbor Discovery

IPv6 nodes on the same link use Neighbor Discovery to discover each other's presence, to determine each other's link-layer addresses, to find routers, and to maintain reachability information about the paths to active neighbors. [18] Neighbor Discovery is the IPv6 equivalent of ARP, which maps layer 2 MAC addresses to IP addresses in an IPv4, network and is a required function for proper communication in the IPv6 protocol which is done automatically by each host interface. Hosts also use Neighbor Discovery to find neighboring routers that are willing to forward packets on their behalf and to actively keep track of which neighbors are reachable and which are not to assist in efficient routing.

Router Solicitation

The purpose of router solicitation is to force routers to generate router advertisements immediately rather than at their next scheduled time. It is used when a node joins the network, and needs to be configured [13]. This message is sent to the all-routers multicast address (`FF01:0:0:0:0:0:0:2` [19]) and has a hop limit value of 255. Figure 2.8 displays the Router Solicitation message format.

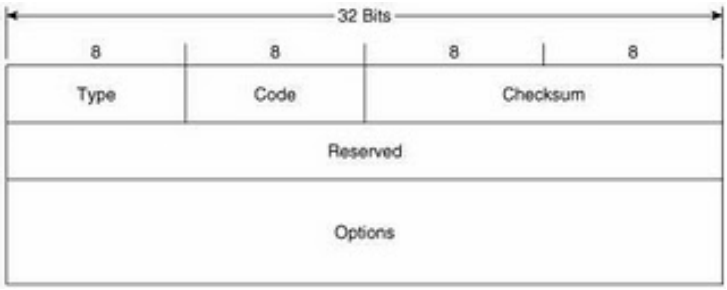

Figure 2.8: Router Solicitation message.

Router Advertisement

Router Advertisements are sent periodically and in response to a Router Solicitation. The source address must be the link-local address of the corresponding router interface (the one that sent it). The destination address can either the source address of an invoking router solicitation or the

the all-nodes multicast address (FF01:0:0:0:0:0:0:1 [19]). The format of the IPv6 Router Advertisement message is shown in Figure 2.9.

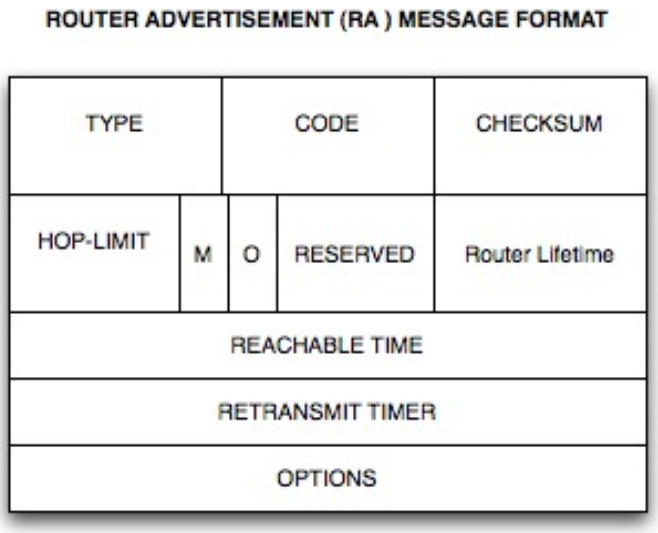

Figure 2.9: Router Advertisement message.

This is an ICMPv6 message with the type number of 134, while the code is 0. The M and O bits are the "managed address configuration" and the "other stateful configuration" flags, which determine whether or not DHCP will be used. The router lifetime is in seconds, and a value of zero means the router is not a default router. The reachable time field indicates the time - in milliseconds - that a node assumes a neighbor is reachable after having received a reachability confirmation [13].

Neighbor Solicitation

Neighbor Solicitation is the IPv6 equivalent to IPv4 ARP requests. Nodes send Neighbor Solicitations to request the link-layer address of a target node while also providing their own link-layer address to the target [18]. When a node needs to resolve an address, Neighbor Solicitations are sent via multicast. When the node seeks to verify the reachability of a neighbor, Neighbor Solicitations are sent via unicast. Figure 2.10 shows the format of the Neighbor Solicitation message.

The type of the Neighbor Solicitation message is 135 and the code is 0. The Source Address is either an address assigned to the interface from which this message is sent or if Duplicate Address Detection (DAD) [20] is in progress the unspecified address(0:0:0:0:0:0:0:0 [14])

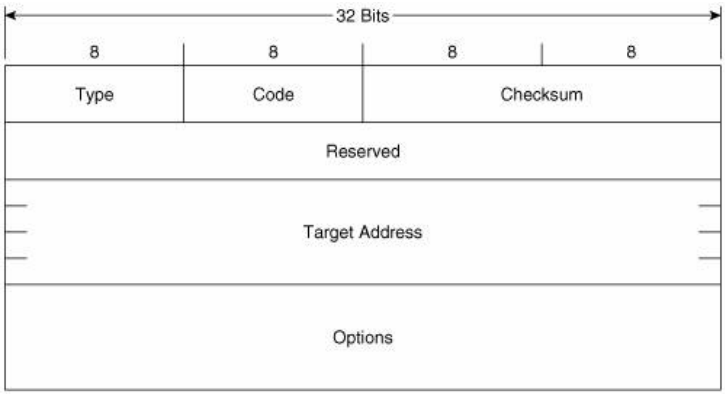

Figure 2.10: Neighbor Solicitation message.

[18].

Neighbor Advertisement

A Neighbor Advertisement is sent out in response to a Neighbor Solicitation or periodically in order to propagate new or changed information quickly. The Neighbor Advertisement message format can be seen in Figure 2.11.

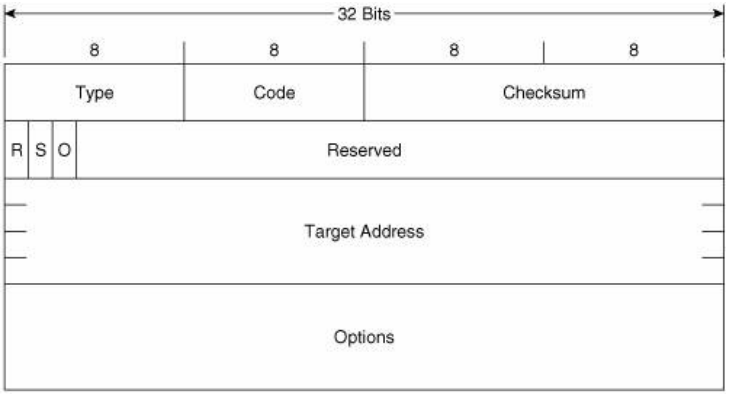

Figure 2.11: Neighbor Advertisement message.

If the solicitation's source address is the unspecified address, the advertisement's destination address is the all-nodes multicast address. The R bit indicates that the node is a router. The S bit indicates that the advertisement is sent as a response to a neighbor solicitation. The O bit is the override flag. When set, it indicates that the advertisement should override an existing cache entry and update the cached link-layer address [13].

Prior Work

In his 2006 thesis, Savvas Chozos tested the security features of IPv6's stateless auto-configuration process as well as the Neighbor Discovory Protocol, both using ICMPv6. During his research Chozos used Jpcap [21] to capture and build appropriate ICMPv6 auto-configuration messages which he then used to implement two DoS threats to the IPv6 auto-configuration procedure in a laboratory IPv6 network. Since the auto-configuration features of IPv6 are focused on user convenience, there are known trade-offs between convenience and security. The result of both of Chozos' DOS attacks were successful. These results led to the conclusion that IPv6 auto-configuration in environments with non-trustworthy hosts is prone to attacks, and if host authentication cannot be achieved, transition to stateful auto-configuration with the use of trusted DHCPv6 servers should be considered. While the most serious effects of the attacks were anticipated, a couple of unanticipated compliance defects in the IPv6 implementation for Linux were also identified. For Chozos, these results indicate that these threats are real, and further studies are required to identify suitable countermeasures [13].

2.2 NIDS Overview

Intrusion Detection is a fundamental practice in Network Security. Network Administrators use many methods and devices to keep the information residing on their networks safe. Firewalls (packet filters) are one such device that can be used to filter out packets that may potentially harm hosts inside a network. A Firewall is a combination of hardware and software that isolates an organization's internal network from the Internet at large, allowing some packets to pass and blocking others. A firewall allows a network administrator to control access between the outside world and resources within the administered network by managing the traffic to and from these resources [22]. Unfortunately, for Intrusion Detection, this is not enough. For effective network security we need a device that not only examines the headers of all packets passing through it (like a packet filter or Firewall), but also performs payload inspection, which is not performed by a packet filter. When this type of device observes a suspicious packet, or a suspicious series of packets, it could prevent those packets from entering the internal network or, because the activity is only deemed as suspicious, the device could let the packets pass, but send alerts to a network administrator, who can then take a closer look at the traffic and take appropriate actions. [22] An NIDS is the type of device that generates alerts when it observes potentially malicious or suspicious traffic. A device that blocks such suspicious traffic is called an intrusion prevention system (IPS), however the IPS will not be covered in this research.

NIDS systems are broadly classified as either signature-based systems or anomaly-based systems. A signature-based NIDS maintains an extensive database of attack signatures. Each signature is a set of rules pertaining to an intrusion activity. A signature may simply be a list of characteristics about a single packet (e.g., source and destination port numbers, protocol type, and a specific string of bits in the packet payload), or may relate to a series of packets [22]. As network traffic flows a signature-based NIDS inspects every passing packet, comparing each packet with the signatures in its database. If a packet matches a signature in the database, the NIDS generates an alert. The Signature-base NIDS has some disadvantages. It requires previous knowledge of an attack in order to generate an accurate signature, leaving it unable to detect new attacks. It may flag suspicious activity that is not actually an attack resulting, in a false alarm, and since every packet must be compared with an extensive collection of signatures. An NIDS can become overwhelmed with processing and actually fail to detect malicious packets.

The anomaly-based NIDS creates a baseline traffic profile as it observes it in normal operation. The NIDS then looks for packet streams that are statistically unusual, for example, an inordinate percentage of ICMP packets or a sudden exponential growth in port scans and ping sweeps [22]. This does not require any previous knowledge other than the baseline traffic profile making it possible to detect new or undocumented attacks. However this advantage is somewhat outweighed by the fact that it is an extremely challenging problem to reliably and consistently distinguish between normal and statistically unusual traffic. Currently most NIDS deployments are primarily signature-based, although some include some anomaly-based features [22].

An NIDS can be used to detect a wide range of attacks, including network mapping, port scans, TCP stack scans, DoS bandwidth-flooding attacks, worms, viruses, OS vulnerability attacks, as well as application vulnerability attacks. Today, thousands of organizations employ NIDS systems [22]. Many of these deployed systems are proprietary, marketed by Cisco, Check Point, and other security equipment vendors. But many of the deployed NIDS systems are public-domain (or open source) systems, such as the popular SNORT [23] and BRO [24] NIDSs. Payload inspection systems like SNORT and BRO utilize regular expressions for their rules due to their high expressibility and compactness. [25] An example of a simple network setup using SNORT is shown in Figure 2.12

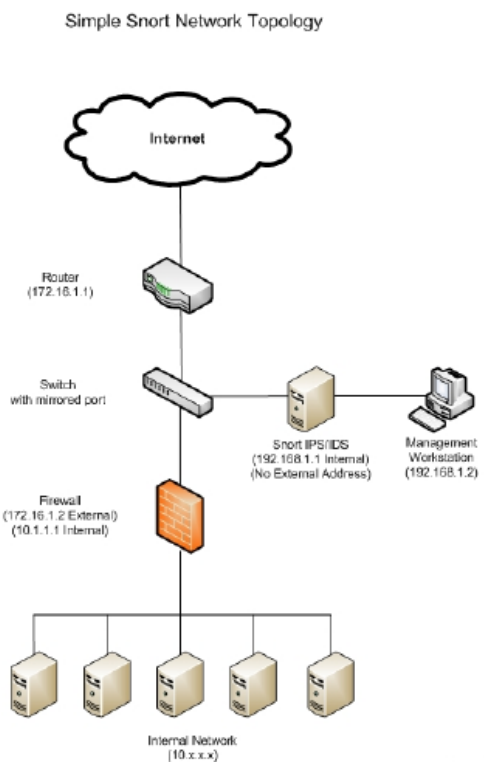

Figure 2.12: Simple example using SNORT NIDS. From [23]

SNORT NIDS

Snort is an open source network intrusion detection/prevention system (NIDS/IPS) developed by Sourcefire. Combining the benefits of signature, and anomaly-based inspection, Snort is the most widely deployed NIDS/IPS technology worldwide [23].

The SNORT NIDS uses the PCRE Engine for regular expression matching on the payload. The software based PCRE Engine utilizes a Non-Deterministic Finite Automata (NFA) engine based on certain opcodes which are determined by the regular expression operators in a rule. Each rule in the SNORT ruleset is translated by a PCRE compiler into a unique regular expression. Since the software based PCRE engine can match the payload with a single regular expression at a time, and needs to do so for multiple rules in the ruleset, the throughput of the SNORT NIDS system dwindles as each packet is processed through a multitude of regular expressions [25].

A large default set of rules comes with SNORT, while also allowing for development of local site rules for customization. SNORT performs protocol analysis, content searching/matching, and can be used to passively detect a variety of attacks, such as buffer overflows, stealth port scans, web application attacks, and OS fingerprinting attempts, as well as many others.

Figure 2.13 illustrates the functional diagram of SNORT followed by a detailed list to describe each function.

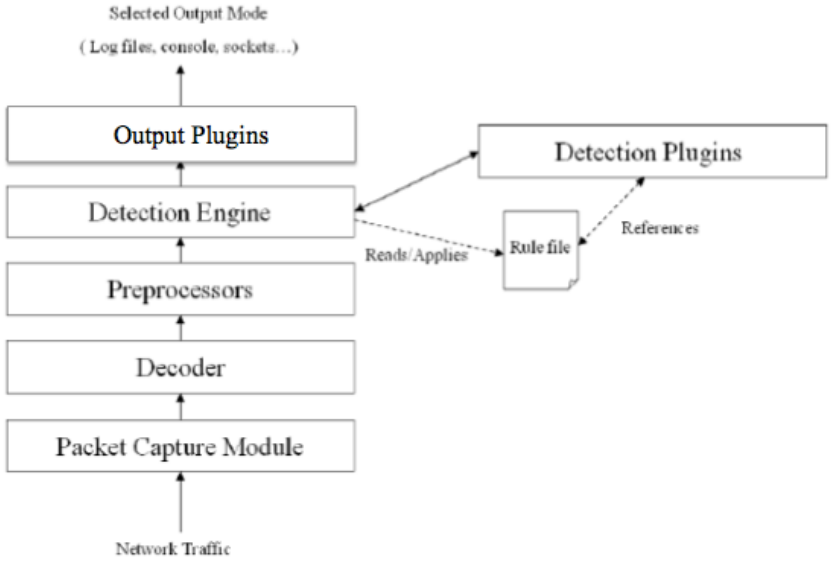

Figure 2.13: SNORT functional diagram from [26]

The following is description of each module is taken from [26].

- **Packet Capture Module** is based on the popular packet programming library libpcap; it can be optimized for performance and provides a high-level interface for packet capture.
- **Decoder** fits the captured packets into data structures and identifies link level protocols. It then looks at the next level, decodes IP, and finally looks at the TCP/UDP in order to determine ports and addresses. SNORT is capable of alerting if it finds malformed headers (unusual length TCP options, etc.)
- **Preprocessors** can be considered a kind of filter, which identifies things that should be checked later (in the next modules e.g., the Detection Engine), such as suspicious connection attempts to some TCP/UDP ports or too many TCP SYN packets sent in a short period of time (port scan). [The Preprocessor's function is to take packets potentially dangerous for the detection engine and try to find known patterns, and thus perform stateful protocol analysis.]

- **Rules Files** are plain text files which contain a list of rules with a standardized syntax. This syntax covers protocols, addresses, and associated output plug-ins. These rules files can be updated locally.

- **Detection Plug-ins** are modules referenced from its definition in the rules files. They are used to identify patterns whenever a rule is evaluated.

- **Detection Engine** The Detection Engine makes use of the detection plug-ins. It matches packets against rules loaded into memory during SNORT initialization.

- **Output Plug-ins** are the modules which allow for the formatting of notifications (alerts, logs) for the user to access them in many ways (console, external files, databases, etc).

BRO NIDS

BRO is an intrusion detection system that works by passively watching traffic seen on a network link. It is built around an event engine that pieces network packets into events that reflect different types of activity. This activity can be anything from seeing a connection attempt, an FTP request or reply, or a user having successfully authenticated during a login session. Bro runs the events produced by the event engine through a policy script, which the BRO administrator supplies, however in general the NIDS will operate using large portions of the scripts (analyzers) that come with the BRO distribution [24].

BRO has its own specialized policy language that allows a site to tailor BROs operation, both as site policies evolve and as new attacks are discovered. If BRO detects something of interest, it can be instructed to either generate a log entry, alert the operator in real-time, or execute a command (e.g., to terminate a connection or block a malicious host). In addition, BROs detailed log files can be particularly useful for forensics.

It should be noted that BRO is intended for use by sites requiring flexible, highly customizable intrusion detection. It has been developed primarily as a research platform for intrusion detection and traffic analysis. It is not intended for someone seeking an “out of the box” solution.

Figure 2.14 displays the functional diagram of BRO.

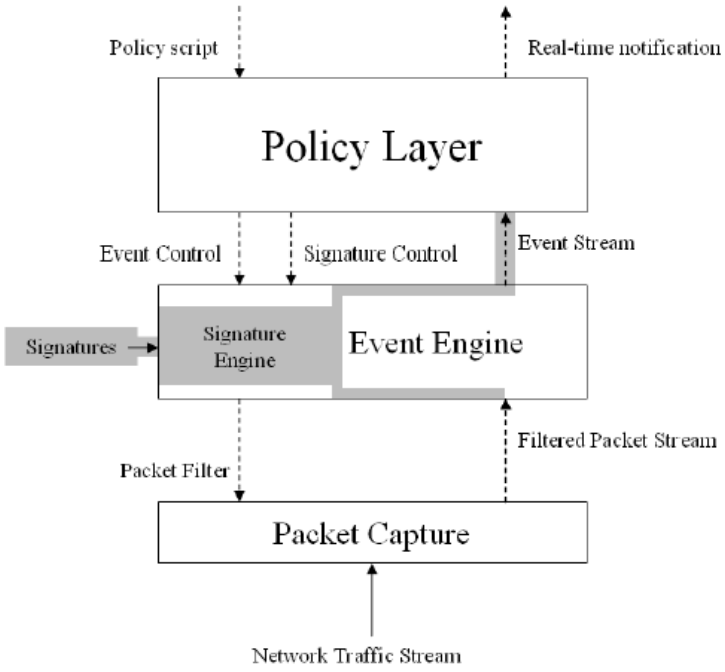

Figure 2.14: BRO functional diagram from [27]

Below is a quick description of what each module does, taken from [28]:

- **Packet Capture** like SNORT, BRO captures traffic using libpcap. This aids in porting BRO to different Unix varieties, and allows it to operate on tcpdump [29] packet traces (for offline analyses).
- **Event Engine** This layer performs several integrity checks to assure that the packet headers are well-formed. For example, it verifies the IP header checksum is correct. At this point BRO reassembles IP fragments so the network layer analyzer can inspect complete IP datagrams. It sends events to the Policy Layer.
- **Signature Engine** inspects the packet stream, and generates an event each time a signature is matched. Those events can then be analyzed by a policy script.
- **Policy Layer** The policy script interpreter executes scripts written in a specialized BRO language. These scripts specify event handlers which are happenings received for the Event Engine Layer, each occurrence is passed to the interpreter.

For high performance, BRO relies on use of an efficient packet filter to capture only a subset of the traffic that transits the link it monitors. Scripts (called analyzers) for analyzing different protocols and different types of activity have been built into BRO for flexibility. This gives the administrator the ability to pick and choose among these analyzers for which types of analysis he/she wants to enable; then BRO will only capture traffic related to the analyzers that have been selected. Overall BRO provides a flexible, highly stateful, efficient, and adaptable network security monitor [24].

2.3 IPv6 vs. NIDS

On 8 June, 2011, top websites and Internet service providers around the world, including Google, Facebook, Yahoo, Akamai and Limelight Networks joined together with more than 1000 other participating websites in World IPv6 Day for a successful global-scale trial of the new Internet Protocol, IPv6 [30]. The collaborators of this event coordinated a 24-hour test flight of IPv6. This event helped demonstrate that major websites around the world are well-positioned for the move to a global IPv6-enabled Internet, enabling its continued exponential growth [30]. However, even with the depletion of IPv4 addresses and the events on World IPv6 day, the adoption of IPv6 has been slow. The development of NIDSs to support IPv6 is no different. There has been little effort to expand the current set of NIDS (Network Intrusion Detection Systems) to support the IPv6 protocol [25]. This has been due mostly to lack of demand. To date, BRO NIDS and SNORT NIDS have implemented IPv6 into their systems. It can be expected that as demand for IPv6 increases, so will development of NIDSs to support IPv6 protocol.

Another area where slow demand for IPv6 has had an impact is the deployment experience of the industry. This has resulted in the lack of experience in securing IPv6 networks. To this point it has not presented a significant problem due to the limited adoption of IPv6. However, IPv6 is becoming a larger target for hackers, and as it becomes more popular it will continue to grow as a target of attacks. As of now, IPv6 is the "wild west" with wide open spaces and few security obstacles for a hacker to avoid or overcome. The IPv6 environment is already being used for malware, attack propagation, and even as Internet Relay Chat (IRC) channels and back doors for more sophisticated hackers [8]. There are already several DoS attacks and a few IPv6 worms. The 2002 Slapper worm attacked Apache Web servers via TCP port 80, creating copies of itself and spreading to other Apache Web servers randomly. The Slapper worm uses a sophisticated command and control channel that would allow a hacker to send commands to

the infected servers. One of these could send a flood of IPv6 packets toward a victim making it the first worm with any type of IPv6 component to it [8]. Another worm, W32/Sdbot-VJ, uses IPv6 to disguise itself [8]. By creating a file (WIPv6.exe) the W32/Sdbot-VJ worm leverages the popularity of IPv6 which would hopefully prevent the user or system from deleting it due to the possibility that it may have something to do with the Windows IPv6 drivers. Several DoS attacks also exist as the hacker community has begun to explore IPv6. This exploration will only increase in its allure as the number of targets increases and IPv6 protocol weaknesses become more understood. In fact, IPv6 capabilities have started to be added to several popular hacker tools. Tools like SCAPY6 [31] and The Hackers Choice (THC) IPv6 Attack Tool Kit [32], both of which are easy to install and relatively simple to use. It is because of this lack of experience in securing IPv6 networks, the growing knowledge of the protocol by attackers, and the current status of the mostly un-monitored environment and tools to take advantage of it, that more hackers will be drawn to IPv6. All of which further emphasize the need for effective NIDSs.

One of the benefits that IPv6 presents to the NIDS is its simplified header and its use of 4 or 8 byte data boundaries. From an NIDS perspective this is excellent because for a modern CPU, taking apart the IPv4 header to detect subtle packet crafting is very inefficient due to the alignment of the data fields. With IPv6, the decomposition of the various fields of the header and extension headers can be done efficiently [25]. This results in a gain in NIDS performance, represented in a per-packet processing speed increase. On the downside of the shift to IPv6 is the possibility that many, many more devices could be connected to the Internet due to the large address space of IPv6 and the technological advances of everything from game systems to refrigerators. The result would be for all of those devices to be exploited and used in an attack. Thus presenting the need to protect or defend these systems with an NIDS. To do this, perhaps the most important hurdle is that of authentication and encryption: a sophisticated NIDS would want to at least verify the validity of the Authentication Header (AH) in each packet, if not check the contents of the Encapsulating Security Protocol (ESP) Header [25]. This presents a problem to performance since it is processing-intensive to decrypt and process these packets on the fly at megabits per second, or higher, rates.

In terms of technology, with the exception of the larger address set and integrated cryptography, there is little else to differentiate IPv6 from IPv4 from an NIDS perspective. The uses of IPv6 technology present the greatest challenges as IPv6 may finally do what IPv4 could not, put "everything on the Internet."

Deployment of IPv6 places increased pressure on the requirement for a paradigm change from current localized solutions to a much more distributed system, and in particular, from the "anti-virus lookalike" to a system finally resembling a proper sentry [25].

2.4 Lists of Exploits Overview

The adoption of IPv6 presents new and unique challenges that industry is just beginning to adapt to. To be able to transition an organization's networks and hosts there are many obstacles to overcome. Perhaps the greatest of these obstacles is security. Although IPv6 both simplifies and improves IPv4, it poses several significant security challenges [33]. The first challenge rests within a built-in feature designed to improve security: IPSec [9]. The problem is that even though IPSec support is mandatory, its use is not, which keeps holes open for old or existing attacks while opening the door for new IPv6 specific ones. To complicate matters, some of IPv6's beneficial features have their own security implications that are not yet fully understood [33]. In addition, not all of the old security problems encountered in IPv4 went away. In fact some problems that affect IPv4 networks, such as application layer attacks, rogue devices, and packet flooding can also affect IPv6 networks [33]. Not to mention any new unanticipated attacks that will be developed once IPv6 has been targeted by the hacking community.

IPv6 security is in many ways the same as IPv4 security [34]. Packets travel through the network via the same mechanisms and any upper or lower layer protocol remains unchanged. Having said that, there are still some significant differences between IPv4 and IPv6 that change the types of attacks that will be seen on an IPv6 network. There has been very little research into what attacks IPv6 networks will face or what mechanisms will be required to protect them. Particular items of concern that need to be addressed are securing Router Advertisements, handling of fragment reassembly and analysis, the lack of Neighbor Discovery §2.1.4 (e.g., ICMPv6 type135/146 messages) and DHCPv6 inspection in edge switches, as well as IPv6 Node Information Queries [35] [36].

Presence of the IPv6 protocol brings new demands for typical network protection mechanisms such as firewalls and intrusion detection systems that need to be upgraded to support IPv6 correctly [7]. The adoption of IPv6 presents a challenge for any mechanism relied upon to secure networks. IPv6 may not only be vulnerable to existing IPv4 threats, but also presents new threats specific to IPv6. The good news is that IPv6 is more resistant to some threats than IPv4. However this does not change that fact that there is significant work to be done in the area of IPv6 security.

General Attack Classes

There are thirteen general attack classes or categories, derived from Convery and Miller's 2004 paper [34], that attackers use to exploit IPv4 networks and hosts. These classes remain relevant with the IPv6 protocol stack. Each of the following general attack classes have either been made significantly easier, harder, or have no impact (or remain consistent) when considered in an IPv6 environment and will be covered in more detail in the following section.

1. Sniffing
2. Application Layer Attacks
3. Rogue Devices
4. Man In the Middle (MITM)
5. Flooding
6. Reconnaissance
7. Unauthorized access
8. Header Manipulation and fragmentation
9. Layer 3 spoofing
10. Address Resolution Protocol (ARP) and Dynamic Host Configuration Protocol (DHCP) attacks
11. Broadcast Amplification Attacks (smurf)
12. Viruses and Worms
13. Translation, Transition, and Tunneling

2.4.1 Consistant Threats

As stated in §2.4, some threats like application layer attacks remain mostly the same in IPv6 as they did in IPv4. This is because the implementation of IPv6 only affects layer 3 and has no direct impact on layer 7 of the OSI model. In fact, many of today's common attacks are application layer attacks. To this group of attacks belong buffer overflow attacks, web application

attacks (e.g., CGI attacks), different types of viruses and worms, etc [7]. Rogue device attacks such as an unauthorized laptop, rogue router, or rogue wireless access point are common in an IPv4 network and are not substantially changed in IPv6 [34]. Transition to IPv6 will not impact these types of attacks.

Flooding Attacks

A flooding attack is a very frequent type of attack. The flooding is an attack that floods a network device, such as a router or a host, with large amounts (more that it can process) of network traffic. This attack can take the form of a local or distributed Denial of Service (DoS) attack and can cause network resources to become unavailable. Arrival of IPv6 did not change basic principles of a flooding attack [7]. However, with the introduction to new extension headers and ICMPv6 message types along with the dependence on multicast addresses, IPv6 may introduce more ways of developing flooding attacks for malicious purposes.

Sniffing

Sniffing or eavesdropping on network traffic also remains unchanged. The tcpdump [29] tool has been implemented with IPv6 support, thus the sniffing of network traffic remains unchanged between IPv4 and IPv6.

Worms

Worms are another threat that remain mostly unchanged when considered with IPv6. Since worms are generally application layer threats, worms such as Melissa which spreads via email will be unaffected in an IPv6 environment [37]. However one of the popular mechanisms for worm propagation is random address-space probing. This allows a fast operating worm to scan an entire address space in a matter of hours. This was emphasized by Staniford *et al.* in their 2002 paper titled "How to own the Internet in your spare time [38]." In this paper, they conclude that it is realistic for an attacker to gain control of a million or more hosts via the use of Worms on an IPv4 Internet. Thus giving the attacker the ability to conduct a mass Distributed Denial of Service (DDoS) attack, access sensitive information (e.g., credit card numbers, passwords), purposely sow confusion, as well as do deliberate damage to infrastructure. In a IPv4 32 bit address space, random address-space probing can be done quickly in order to find new hosts to infect. However when attempted with IPv6, with its 128 bit address space, scanning is a much more difficult and time consuming task. In fact, if we assume that the number of hosts on the Internet does not increase by a factor proportional to the address increase, then the work factor for finding a target in an IPv6 internet would be approximately 2^{96} greater than that of

IPv4 [37]. This work increase would seem to make the random scanning worms irrelevant due to the scan time expense.

In the Bellovin *et al.* paper analyzing propagation strategies of worms in an IPv6 environment, they noted that address-space scanning worms such as Code Red will have a tough time effectively finding vulnerable victim hosts in an IPv6 environment [37]. The adoption of IPv6 removes one of the two ways address-space scanning is currently performed by making it infeasible to use a uniformly distributed random number generator to select new target addresses [37]. The other method, biasing the search space by scanning within the same subnet, preferentially spreads locally and is a much more feasible method [37]. However even scanning within an IPv6 subnetwork seems to be an unattainable goal for any worm. Even with only having to scan the local IPv6 subnet, a worm would be required to scan through 80 bits of local address space, which is a massive number and daunting obstacle. Thus, in an IPv6 environment, the worm threat itself does not change. However the method by which worms find hosts may have to change.

Man In The Middle (MITM)

The general theory of the Man in the Middle (MITM) threat does not change with IPv6. Because the IPv4 and the IPv6 headers have no security mechanisms themselves, each protocol relies on the IPsec protocol suite for security [34]. Since it is only mandatory for IPv6 implementations to *support* IPsec, but does not require it to be used, IPv6 falls prey to the same security risks posed by a MITM attack.

Table 2.2 lists the IPv4 attacks that remain mostly unchanged when converted to IPv6. The table shows current IPv4 attack types or vulnerabilities next to their IPv6 counterpart.

Attacks Mostly Unchanged When Converted to IPv6		
Attack Type		
	IPv4 Attack	**Analog IPv6 Vulnerability**
Sniffing		
	TCPDUMP	TCPDUMP(IPv6 enabled)
Man in The Middle (MITM)		
	Various	Parasite6
	Various	Redirect6
Flooding		
		Flood_Advertise6
		Fake_router6

Table 2.2: Unchanged attacks

2.4.2 New Threats

This section outlines attacks that change significantly when considered in the IPv6 protocol address space. This section is based largely on Convery and Miller's 2004 paper on threat comparison and best practices [34]. IPv6 specific threat studies will also be covered in this section, however this will be as they relate to host and not NIDS detection.

Reconnaissance

Generally the first attack performed, reconnaissance, is an attempt by the adversary to learn about your network in an effort to find possible holes or weaknesses. Convery and Miller state, "this includes both active network methods such as scanning as well as more passive data mining such as through search engines or public documents. [34]" The active host probing or port scanning is an attempt for an attacker to discover specific information about hosts and network devices on the victim's network. This includes how they interconnect and what traffic is being passed between them. Passive data mining can be considered environmental data to assist the attacker in theorizing different ways to attack the victim network.

Typical IPv4 methods of collecting information are ping sweeps, port scans, and application and vulnerability scans. Reconnaissance in IPv6 differs from IPv4 in two relevant ways. The first is that ping sweep or port scan, when used to enumerate hosts on a subnet, are much more difficult to complete in an IPv6 network [34]. This is emphasized by Caicedo et al. in their 2009 paper where they note, the potentially huge size of IPv6 subnets makes reconnaissance attacks more

difficult [33]. The large size of the IPv6 address space makes port scanning, whose procedures are identical for IPv4 and IPv6, more tedious and time consuming. With a default subnet on an IPv6 network being 64 bits, that means that to perform a scan on the whole subnet it is necessary to make 2^{64} probes, a next to impossible task [7]. Or as summarized by Convery and Miller, a network that ordinarily required sending 256 probes now requires sending more than 18 quintillion probes to cover an entire subnet. A task that would take "28 centuries of constant 1-million-packets-per-second scanning" to find the first host on the first subnet on a /64 IPv6 network containing 100 active hosts [34].

In an attempt to prove application vulnerability scans, or OS fingerprinting, have similar possibilities in IPv6 as they do in IPv4. Nerakis, in his 2006 thesis, used existing TCP/UDP packet probing methods along with IPv6 Extension Headers to attempt to determine the version and type of remote host operating systems. Nerakis discovered that existing TCP/UDP methods work, however it is more difficult to perform in an IPv6 environment. This is believed to be due in part to the larger address space and the (at the time) experimental nature of IPv6 with similar OSs possibly reusing IPv6 code [39].

IPv6's reliance on Multicast addresses will do just the opposite, making the adversaries life easier. The multicast address structure as defined in RFC 2375 [40] lets the attacker identify groups of key network components, such as the all router or all DHCP servers for a given network. This gives the attacker an almost hand delivered list of devices to scan for vulnerabilities, making his reconnaissance possible if not easy. This defined list of multicast addresses is clearly for legitimate protocol use, however it opens IPv6 up for reconnaissance as well as "simple flooding" attacks, or something more sophisticated that is designed to subvert the device [34].

Unauthorized Access

The Unauthorized Access attack is the type of attack in which an adversary tries to exploit the open transport policy found in IPv4. There is nothing in the protocol stack that limits the number of hosts that can connect to one another on an IP network. Attackers rely on this fact to establish connectivity to upper-layer protocols and applications on inter-networking devices and hosts [34].

To aid in preventing this attack the need for access controls is the same in IPv6 as it is in IPv4, though eventually the requirement and use of IPsec may enable easier host access control. Besides the mandatory support for IPsec, IPv6 technology differences that enable unauthorized access include Extension Headers, ICMP, Multicast and Anycast Inspection. In the case of Extension headers, which replaced the IPv4 IP options, all IPv6 endpoints are required to accept IPv6 packets with a routing header. This can be used by an attacker to circumvent security policies because of the possibility that the endpoint does not only accept the IPv6 packet but also processes it and forwards it, which possibly bypasses a networks firewalls [34]. For ICMP, which is not required by IPv4, current best practices are generally in favor of complete filtering. However, ICMPv6 is an integral part of IPv6 operations and cannot be completely filtered without preventing communications. Since many of the utilities in IPv6, such as Neighbor Discovery, use ICMPv6, there are many opportunities to use it to aid in an Unauthorized Access Attack and thus subvert the networks' security policies or bypass firewalls.

Header Manipulation and Fragmentation

The misuse of IPv6 routing and fragment headers can give an adversary tools to perform DoS attacks as well as avoid access controls. RFC 2460 [1] stipulates that all IPv6 nodes have to be able process routing headers. Because routing headers can be used to avoid access controls based on destination addresses, this presents a significant security issue. For instance, if an intruder sends a packet to a publicly accessible address with a routing header containing a "forbidden" address (address on the victim network) the publicly accessible host will forward the packet to a destination address stated in the routing header (e.g.,"forbidden" address) even though that destination address is filtered [7]. This enables the adversary the ability to perform a DoS attack by using this publicly accessible host to redirect spoofed packets.

Historically fragmentation has been used in IPv4 to bypass access controls or slip attacks past routers, firewalls, and in particular NIDSs. The IPv6 specification [1] does not allow packet fragmentation by intermediary devices. In other words, only the source host can perform fragmentation. The source node could move port numbers from the first fragment to bypass security monitoring devices (which do not reassemble fragments) expecting to find transport layer protocol data in the first fragment [7]. Convery and Miller hypothesize that it is also possible to use the combination of multiple extension headers and fragmentation to create the same ability for the intruder to hide an attack. Since the "IPv6 minimum MTU is 1280 octets" a good security policy for IPv6 would have any packet with a MTU less that 1280 be dropped, but this is not always the case. Thus, an attacker can use large numbers of packets with small fragments

(MTU $<$1280) to overload reconstruction buffers on the target system in an attempted DoS and possibly causing it to crash [34] [7] .

Layer 3 Spoofing

"Spoofing" is the ability of an attacker to modify their source IP address and the ports they are communicating on to appear as though traffic was initiated from another location or application. In the current IPv4 environment these attacks occur everyday, enabling adversaries to perform DoS attacks or conduct spam, worm, or virus attacks. A promising benefit of the IPv6 protocol is the globally aggregated nature of the IPv6 address space. This means that IPv6 allocations are set up in such a way as to easily be summarized at different points in the network. Thus, Internet Service Providers can perform network egress filtering [41] to ensure that at least their customers are not spoofing outside their own ranges. This, unfortunately, is not required and has little impact on "spoofing" attacks; keeping this type of attack as relevant in IPv6 as it was in IPv4 [34].

Address Resolution Protocol (ARP) and Dynamic Host Configuration Protocol (DHCP) attacks

ARP and DHCP attacks attempt to subvert the host initialization process or a device that a host accesses for transit. These types of attacks try to get end hosts to communicate with an unauthorized or compromised device or attempt to configure these hosts with incorrect network information such as default gateway, DNS, or IP address [34].

In their 2004 paper, Convery and Miller note that IPv6 has no inherent security added to the IPv6 equivalents of DHCP (DHCPv6) or ARP (Neighbor Discovery). With the preference to Stateless Auto-configuration over DHCP, IPv6 is open for attacks in these areas. Although DHCP servers may be used on occasion, especially for setting up hosts with network configuration information, dedicated servers are not common in IPv6. Stateless Auto-configuration messages can be spoofed to allow an adversary to deny access to devices [34].

With the many types of devices that now connect to todays networks and the Internet, IPv6 replaces ARP with elements of ICMPv6 called Neighbor Discovery §2.1.4, which has the same inherent security as ARP in IPv4 [34]. This means that there are many options for these types of attacks in IPv6 based on vulnerabilities associated with ICMPv6. IPsec, the default security mechanism for IPv6, does not allow for automatic protection of the auto-configuration process. Thus, the Secure Neighbor Discovery Protocol (SeND) [42] was created to protect this process.

SeND uses Cryptographically Generated Addresses (CGA) and asymmetric cryptography as a first line of defense against attacks on integrity and identity.

In his 2007 thesis, Marcin Pohl evaluated SeND. He found that even thought SeND claims to achieve mutual authentication of hosts and routers without the need for a Certification Authority (CA), SeND does not really offer mutual authentication without a CA and is susceptible to CPU exhaustion attacks [43]. However, without SeND both router and Neighbor solicitation and advertisement messages can be "spoofed" and will overwrite existing neighbor discovery cache information on a device, resulting in the same issues present in IPv4 ARP. What this means is that a spoofed router discovery could inject a bogus router address that hosts listen to and perhaps choose for their default gateway. Then the bogus router can record traffic and forward it to proper routers without detection; leaving the adversary the ability to perform MITM or DoS attacks at will [34].

Ciacedo *et al.* cover a DoS attack on the Duplicate Address Detection (DAD) protocol and its procedures in their 2009 paper. In this type of attack an attacker on the local link waits until a node sends an NS packet. The attacker then falsely responds with a neighbor advertisement (NA) packet, informing the new node that it is already using that address. Upon receiving the NA, the new node generates another address and repeats the DAD process; the attacker again falsely responds with an NA packet, thus repeating the whole process. They note, eventually the new node gives up without initializing its interface [33] .

Another possible attack exploiting the stateless auto-configuration process is a MITM attack. This is possible when a node needs a MAC address of another node on the subnet and sends a NS message to the all-nodes multicast address. An attacker on the same link can see the NS message and reply to it with the corresponding NA message, thereby taking over all traffic between the two original nodes.

Figure 2.15 graphically shows the process of three IPv6 auto-configuration (ARP/DHCP) type attacks.

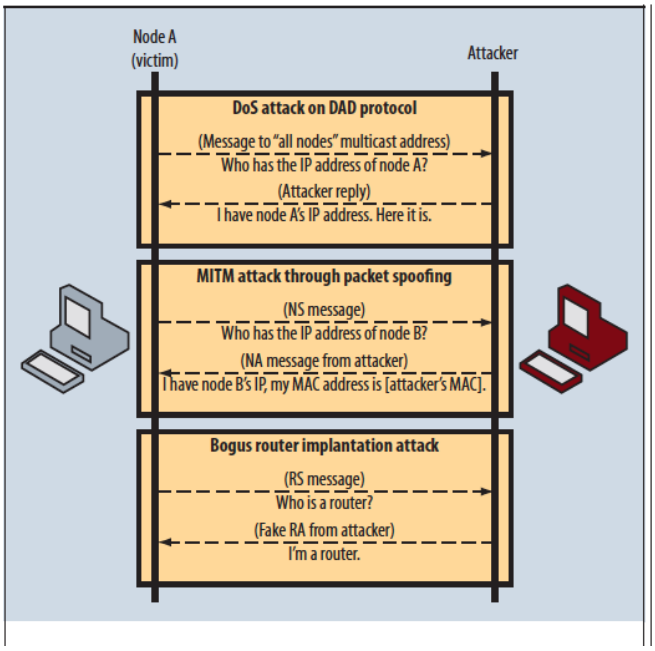

Figure 2.15: Attacks on IPv6 related to the auto-configuration process. From [33]

Broadcast Amplification Attacks (smurf)

Broadcast Amplification attacks, also know as "smurf" attacks, are DoS attacks the take advantage of the ability to send echo requests with a destination address of a subnet broadcast and a spoofed source address, using a victim's IP address. This generates a response from all host on that subnet directly to the victim's host creating a flood of echo response messages.

In IPv6, since the concept of broadcast addresses is removed from the specification [1] and protocol, these types of attacks are mitigated. In regard specifically to "smurf" attacks, ICMPv6 messages should not be generated as response to a packet with a multicast destination address, a link layer multicast, or a link layer broadcast address as stated in RFC 2463 [44]. Even though this effectively kills "smurf" attacks in properly implemented IPv6 stacks, exceptions are made making these attacks possible on the local subnet. If a target has mis-implemented IPv6, it responds with an echo reply to the All-Nodes multicast address, this generates a mass of response traffic sent directly to the target. However, if each host has a properly implemented IPv6 stack, these attacks are effectively mitigated.

Viruses and Worms

As discussed in §2.4.1, traditional worms and viruses remain unchanged with IPv6. The propagation methods of these types of attacks may encounter some difficulty with the large address

space, which is seen in the Bellovin *et al.* 2006 paper [37]. Because of this difficulty most worms would be less effective in an IPv6 environment simply because of their inability to find hosts to infect.

Translation, Transition, and Tunneling

With the transition to IPv6 already underway, careful consideration must be given to the period between native IPv4 and native IPv6. As IPv4 networks are converted, there will be a considerable period of time where a transition mechanism will be required. During this "in-between" time, vulnerabilities specific to the transition mechanism must be taken into account and evaluated. There are several approaches to transitioning from IPv4 to IPv6:

- Dual stack
- Tunneling
- Translation

Each of these approaches have their own security considerations to be taken into account when deciding how to transition to IPv6. Convery and Miller state that the existence of so many transition technologies creates a situation in which network designers need to understand the security implications of the transition technologies and select the appropriate one for their network [34]. To this point when discussing IPv6 native access, we have discussed vulnerabilities and attacks that assume the end host is dual stacked, having both IPv4 and IPv6 infrastructure.

Tunneling refers to the transmission of data intended for use only within a private, sometimes corporate, network through a public network in such a way that the routing nodes in the public network are unaware that the transmission is part of a private network. In this case, IPv6 is tunneled through the public IPv4 infrastructure. The IPv6 global Internet uses numerous tunnels over the existing IPv4 infrastructure. This is generally done through a tunnel broker such as Hurricane Electric [45] due to the complexity of setting up and managing these tunnels. Tunnels are difficult to configure and maintain and too complex for the isolated end user, so the concept of the tunnel broker was presented to help early IPv6 adopters to hook up to an existing IPv6 network. Additionally Convery and Miller noted that in many of the transition studies done, automatic tunneling mechanisms are susceptible to packet forgery and DoS attacks. These risks are the same in IPv4, however IPv6 increases the number of paths of exploitation for the adversaries [34]. Relay translation technologies, such as 6to4 [46], introduce automatic

tunneling with third parties as well as additional DoS possibilities. Although, much like the case with tunneling IPv6, new avenues for exploitation are created and the risks do not change from those with IPv4 [34, 46]. Finally, this thesis is not intended to discover new threats or vulnerabilities associated with all of the transition mechanisms, however it will explore the reaction of NIDSs to known IPv6 threats in a tunneled environment.

Table 2.3 lists IPv4 attack classifications that have new or unique considerations when converted to IPV6. Just as in Table 2.2, the table shows current IPv4 attack types or vulnerabilities next to their IPv6 counterpart. Note: Tunneling will be tested on a modified testbed in experiments to be run separately, and all tests assume each host is operating in a dual stack environment.

Attacks with Special Consideration When Converted to IPv6		
Attack Type		
	IPv4 Attack	**Analog IPv6 Vulnerability**
Reconnaissance		
	Ping Sweep	Ping Sweep (Detect-New-IP6)
	Port Scan	Port Scan (Alive 6)
		Multicast (Alive6)
Unathorized Access		
	IP Options	Extention Headers
	ICMP	ICMPv6 (Redir6)
		Multicast Insp(Fake-Advertise6)
		Anycast Insp
Header Manipulation		
and Fragmentation		
	Fragmentation	Overlapping Fragments
		Out of Order Packets (evasion)
Spoofing		
	Layer 3 (Ip Addr)	Spoof6
	Layer 4 (SYN flood)	Syn6
ARP & DHCP		
	DHCP MITM	DHCP6
		SLAAC
		ICMP6 (RA, Discovery)
	ARP	ICMP6
		Nieghbor Discovery (NDP)
		Parasite6(like ARP MITM)
Bcst Amplification (SMURF)		
	SMURF	SMURF6
		RSMURF (linux)
Routing Attacks		
	(Flooding,	OSPFv3, RIPng
	Rapid Announcement,	Flood-Router6
	Route Removal,	Fake_MLDrouter6
	Bogus Routes)	Fake_Router6

Table 2.3: Attacks with new considerations in IPv6

2.5 Fuzz Testing Overview

Fuzz testing or fuzzing is a software testing method that stemmed from a paper written in 1990 by Miller *et al.* [10] The origin of the idea came on "one dark and stormy night," when one of the authors was logged onto his workstation connected via a dial-up line from home and the rain had affected the phone lines; resulting in frequent spurious characters on the line [10]. What they realized was that those spurious and random characters would, at times, crash programs on the workstation. These programs included a significant number of basic operating system utilities [10]. With the assertion that "basic utilities should not crash," they started a systematic test of utility programs running on various version of the UNIX operating system. As a result of their testing they were able to crash 25-33% of the utility programs on any version of UNIX that was tested [10].

Today, fuzz testing is often an automated or semi-automated process, that inputs invalid, unexpected, or random data into a computer program. The program is then monitored for errors or exceptions such as crashes. This method is commonly used to test for security problems in software or computer systems prior to their release or use. For the purpose of this research fuzz testing will be accomplished by generating random inputs to parts of IPv6 packets (e.g., header fields, extension headers, or data) and monitoring each NIDS as well as the destination hosts for symptoms of errors or exceptions. This series of fuzz tests will explore the impact on each NIDS by conducting fuzzing in the areas of ICMPv6, fragmentation (extension headers), multicast, and router advertisement.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 3:
Methodology

The intent of this thesis was to determine: i) which exploits that exist with IPv4 signatures are feasible or infeasible with conversion to IPv6, and ii) will popular open source NIDSs detect these IPv6 attacks, as evidenced by testing two open source NIDSs. We broke down attack types into thirteen general categories, looked for an IPv4 attack in each category, transitioned that attack to IPv6 and then sent that attack in a virtualized network. If the attack was specific to IPv6, we developed that attack or used existing implementations, e.g., [32]. Once the attacks were conducted, an attempt was made to determine the cause of any false negatives. To do this we had to look deeper into each NIDS and study their detection mechanisms and flow.

At the onset of our research our intent was to test each NIDS as an "out of the box" configuration in an attempt to determine the readiness at initial installation. In practice, we found that each of the NIDSs were so locally configurable that their default installation required unique configuration changes, such as Ethernet adapter, ICMP detection profile, and scan detection. This required an initial change in philosophy, as we ended up testing the NIDSs in a slightly modified from default configuration. For example both NIDSs had to be configured with the `--enable-ipv6` or `--enable-brov6` option in order for them to process IPv6 traffic. SNORT also required some changes in the *snort.conf*, **sfPortscan** in particular, which needed uncommenting to enable port scan detection as well as changing the sensitivity to these scans from Low to High. We also had to set the path to SNORT's decoder and preprocessor rules, as well as uncomment `include $PREPROC_RULE_PATH/preprocessor.rules` and `include $PREPROC_RULE_PATH/decoder.rules` in *snort.conf*. All other changes involved network and host ethernet adapter setup variables.

3.1 Test Bed

To effectively test both NIDSs, an IPv6 "Native" (completely IPv6) test bed was constructed, as shown in Figure 3.1. For this, we used a virtual network designed in Oracle's VirtualBox [47] running on a Dell Precision T3500, Dual Intel Xeon 2.4GHz, 6 GB RAM, 2x 500 GB hard drives with Windows 7 SP1 [48] as the host operating system. The virtual machines Test_host, End_host, End_host2, BRO_server, and SNORT_server were each loaded with Ubuntu 10.10 [49]. SNORT_server was configured with SNORT version 2.9.0.5, while BRO_server was

configured with BRO version 1.5.3. Finally BRO2_server was loaded with FreeBSD release 8.2 configured with BRO version 2.0.

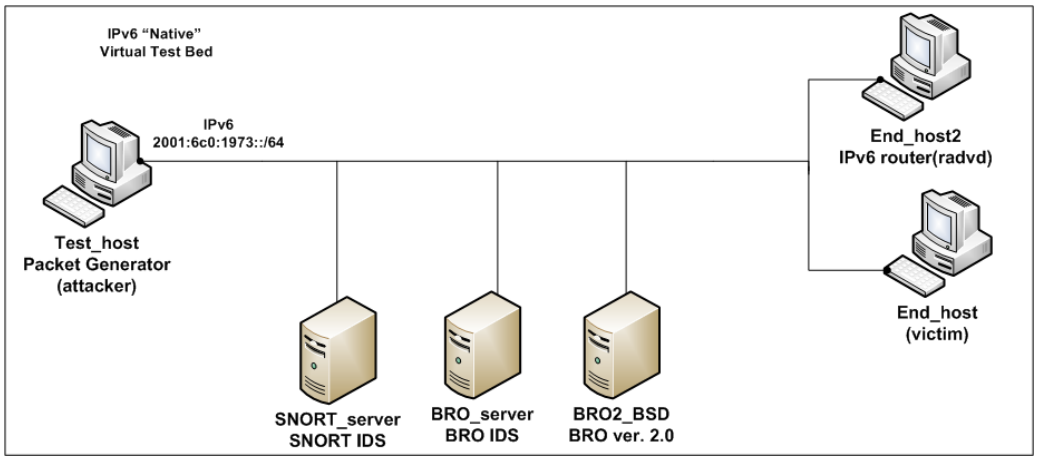

Figure 3.1: IPv6 "Native" test bed.

3.1.1 IPv6 Transitional Test Bed

To appropriately test the NIDSs in a Transitional environment a few changes needed to be made to the "Native" Test Bed outlined in §3.1. Displayed in Figure 3.2, the Transitional Test Bed includes moving End_host2, connecting it via IPv6 to Test_host to allow it to act as a router (radvd) and then connecting it to End_host via a V4 to V6 tunnel. The placement of each NIDS outside this tunnel enabled us to determine reactions to attacks in a Transitional environment.

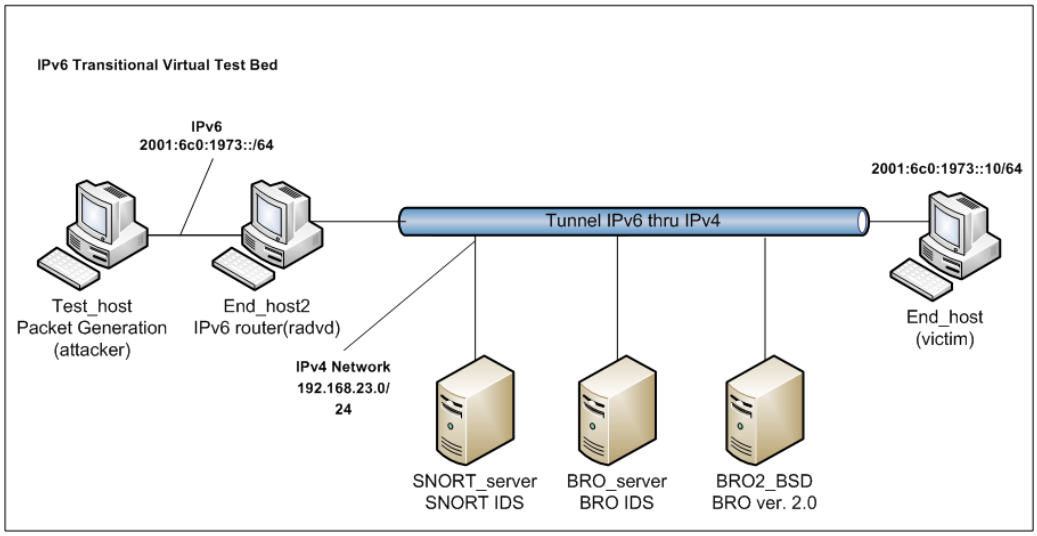

Figure 3.2: IPv6 Transitional test bed.

3.2 Baseline

To ensure initial detection capabilities and NIDS setup, a baseline needed to be created. The easiest way to set this baseline was to find a well known IPv4 attack or detectable reconnaissance method that had a rule or signature built into each NIDS. Thus a baseline was set using two sequential TCP port scans, one written in SCAPY [31] *portscan4.py* and the other using NMAP [50], in IPv4. Each of these scans send packets one right after another as fast as possible in order to scan the target hosts TCP ports. Port scans were chosen as a baseline since each NIDS had a built-in rule/policy for detection of this type of traffic and port scans are fairly simple to construct and use. This also gave us a method that was above IP layer but below the Application layer in order to test the NIDSs operation. After some simple additional configuration changes, each NIDS detected the baseline IPv4 attacks. The next step was to send the corresponding attack using IPv6 while monitoring the network and NIDS logs for activity.

3.3 The Attacks

Table 3.1 lists the general attack classes with IPv4 attack types and their respective IPv6 counterpart. Since our purpose was to test the IPv6 readiness of the chosen NIDSs we used the thirteen general attack classes, then found IPv4 attacks that fit into those classes. From these attacks we either devised corresponding IPv6 attacks in SCAPY6 [31], a Python based packet generation tool, or found existing attacks. Our primary source of existing IPv6 attack implementations was the THC IPv6 Attack-toolkit [32]. Note that Table 3.1 only lists ten general classes. This is due the exclusion of Application layer attacks, the joining of sniffing and reconnaissance into one class, and covering Transition, Translation, and Tunneling in the Transitional test case §3.4.

Attacks to determine NIDS IPv6 Readiness

Attack Type		
	IPv4 Attack	**IPv6**
Reconnaissance		
	Port Scan	Alive6(resolve addresses)
		Alive6(Invalid Header)
		Alive6(Invalid Hop by Hop)
	portscan.py	portscan6.py
	NMAP	NMAP6(scan ports)
		NMAP6 (TCP connection scan)
Unauthorized Access		
	ARP Poisoning(ICMP)	Fake_Advertise6
	DNS	Fake_DNS6
	Route Implanting	Toobig6
Host Configuration		
	DHCP DoS	DoS-new-IPv6
		Detect-new-IPv6
		Fake_DHCPS6
		NDPexhaust
Broadcast Amplification Attacks(SMURF)		
	SMURF	SMURF6
		RSMURF6
Routing Attacks		
	Various	Fake_Router6
		Fake_MLDrouter6(Solicitate)
		Fake_MLDrouter6(Terminate)
		Fake_MLD26(Query)
		Fake_MLD26(Add)
		Fake_MLD26(Delete)
		Kill_router6
Man in The Middle (MITM)		
	Various	Parasite6
Flooding		
	Various DoS	Denial6(large Hop by Hop)
		Denial6(large dst Header)
	DHCP flooding	Flood_DHCPc6
	Various	Flood_Advertise6
		Flood_MLD6
		Flood_MLDrouter6
		Flood_Solicitate6(Network)
		Flood_Solicitate6(Target IP)

Attacks to determine NIDS IPv6 Readiness (cont)		
Attack Type		
	IPv4 Attack	**IPv6**
Rogue Devices		
	Rogue Router	Fake_Router6
		Fake_MLD6
		Fake_MLDrouter6(Advertise)
Fragmentation		
	Fragmentation	Fragmentation6
		Fake_Router6(Fragmentation)
		Kill_Router6(Fragmentation)
Exploit		
	Malware Transfer	FTP Malware
	Various	Exploit6
		Implementation6

Table 3.1: Attack Matrix

Reconnaissance

Generally the first attack performed, as stated in §2.4.2, was reconnaissance; i.e., an attempt by the adversary to learn about a network in an effort to find possible vulnerabilities or weaknesses. Included in this is both active and passive methods. One example of active host probing, or port scanning, is an attempt for an attacker to discover specific information about hosts and network devices on the victims network. This includes how they interconnect and what types of traffic is being passed on the subnet. Passive data mining, which can be considered a method to retrieve environmental data to assist the attacker in theorizing different ways to attack a victim's network, would be a good example of a passive method. Examples include sniffing network traffic, looking at BGP tables in order to determine network addresses, browsing the targets web site, or even "Who-is" searches in order to gain information on the target's IP address space. For our testing, our initial selection for reconnaissance was **Alive6**, however we also included **NMAP** scans, as well as ***portscan6.py***.

Alive6 shows the attacker what IPv6 addresses are alive and working in a given network segment. This is done with a variety of scan types including, Invalid Header, Ping, tcp syn ssh, and Invalid Hop by Hop. Each of these attack options allow an attacker to determine not only how a host reacts to each type of scan sent, but also lets him/her know if there are active hosts on the network segment and what their addresses are. For simplicity, we used the Invalid Header and Invalid Hop by Hop options.

portscan6.py is a simple 1000 port scan starting at port 0, written in SCAPY.

Unauthorized Access

The Unauthorized Access attack §2.4.2 is the type of attack where an adversary tries to exploit the open transport policy, that does not limit the number of hosts that can connect to a one another on a IP network, found in IPv4. Attackers rely on this fact to establish connectivity to upper-layer protocols and applications on inter-networking devices and hosts [34]. To re-iterate a point covered in §2.4.2, besides the mandatory support for IPsec, IPv6 technology differences that enable unauthorized access include extension headers, ICMP, multicast and anycast Inspection. In the case of extension headers, which replaced the IPv4 IP options, all IPv6 endpoints are required to accept IPv6 packets with a routing header. This can be used by an attacker to circumvent security policies because of the possibility that the endpoint not only accepts the IPv6 packet but also processes it and forwards it, which possibly bypasses a network firewalls [34]. For ICMP, which is not required for communications in IPv4, current best practices are generally in favor of complete blocking. In contrast, ICMPv6 is an integral part of IPv6 and cannot be entirely blocked without preventing communications. Since many components of IPv6 use ICMPv6, e.g., Neighbor Discovery, there are many opportunities to use ICMPv6 to aid in an Unauthorized Access attack and thus subvert network security policies or bypass firewalls. For this purpose, **Fake_Advertise6**, **Fake_DNS6d** and **Toobig6** were used in our tests.

Fake_Advertise6 allows the attacker to falsely advertise an IPv6 address using the Neighbor Advertisement process (NA) detailed in §2.1.4. The false NA can either be sent to a specific target or the all-nodes multicast address. **Fake_DNS6d** creates a simple, but fake, DNS server that serves the same IPv6 address (set by the attacker) to any lookup request it receives. While the **Toobig6** attack allows an attacker to set a specified MTU on a target, thus allowing for a possible redirect of traffic and a subsequent unauthorized access. This is the same idea as implanting a route using ICMPv6 redirects, except in this case **Toobig6** uses this method to reduce the MTU of the victim.

Host Configuration Attacks

ARP and DHCP attacks as covered in §2.4.2 attempt to subvert the host initialization process or a device that a host accesses for transit. These types of attacks try to get end hosts to communicate with an unauthorized or compromised device, or attempt to configure these hosts with incorrect network information such as default gateway, DNS, or IP address [34]. For this

type of attack we chose a Denial of Service (DoS) in **DoS-new-IPv6** and **Detect_new_IPv6**, **Fake_DHCPS6** and finally **NDPexhaust**.

Both **DoS-new-IPv6** and **Detect_new_IPv6** take advantage of the SLAAC and NDP §2.1.4 processes to either deny or discover new IPv6 addresses. **DoS-new-IPv6** uses Duplicate Address Detection (DAD) discussed in §2.4.2 in order to conduct a DoS on any host attempting to create a new IPv6 address. This is done by simply sending a message, stating that the address selected by that host is already taken, back to the host attempting to create a new address. **Fake_DHCPS6** is a simple attack that creates a fake DHCP server on the network that can be used to configure a host with an address as well as set a DNS server, allowing the attacker to point hosts to a server of their choosing. The **NDPexhaust** attack continuously pings randomly chosen IPs in the target network. This requires all hosts on that network to complete the NDP process, performing Duplicate Address Detection (DAD) as well as develop a neighbor discovery table in cache, for each ping which eventually starves them of resources.

Broadcast Amplification Attacks (smurf)

Broadcast Amplification attacks, also know as "smurf" attacks §2.4.2, are DoS attacks that take advantage of the ability to send echo requests with a subnet broadcast destination address and a spoofed source address of the victim's IP. This requests generates a response from all hosts on that subnet directly to the victim's host, creating a flood of echo response messages. This was an extremely versatile attack with IPv4, however in IPv6 it was considered to be obsolete due to the lack of broadcast addresses. This is not the case, as attackers *can* take advantage of multicast addresses and mis-implemented IPv6 stacks. This type of attack is still very relavent on the local subnet, where you can generate loads of network traffic, and may also be viable on remote subnets that have not implemented IPv6 correctly (this is a rare exception to the rule). The THC toolkit [32] provided **SMURF6** and **RSMURF6**, both of which were used to represent this type of attack.

Routing Attacks

This class of attack includes any attack focused on network routing mechanisms. For our purpose, we chose **Fake_Router6**, **Fake_MLDrouter6**, **Fake_MLD26**, and **Kill_router6**.

Fake_MLDrouter6 is used for Multicast Listener Discovery (MLD) [51] and allows you to solicit and terminate membership, while **Fake_MLD26** is used for MLDv2 [52] and allows the attacker to query, add, or delete MLD devices.

Man In The Middle (MITM)

The Man in the Middle (MITM) attack involves any attack where an attacker can insert themselves into the network between hosts and monitor or modify traffic without knowledge of the communicating hosts. The general theory of the MITM threat does not change with IPv6. Because the IPv4 and IPv6 headers have no security mechanisms themselves, each protocol relies on the IPsec protocol suite for security and authentication [34]. Since it is only mandatory for IPv6 implementations to *support* IPsec and is not required to be used, IPv6 falls prey to the same security risks posed by a MITM attack. In fact there are plenty of new opportunities for MITM attacks in IPv6 for example an attacker can create fake router using Router Advertisements (RA). For this attack we used **Parasite6**.

Parasite6 is an IPv6 ARP spoofer that redirects all local traffic to the attacker's system by falsely answering Neighbor Solicitation (NS) §2.1.4 requests, allowing the attacker to claim to be any system on the network.

Flooding Attacks

As covered in §2.4.1 a flooding attack is a common attack type. It is an attack that can flood a network device, such as a router or a host, with large amounts (more that it can process) of network traffic. This attack can take the form of a local or distributed Denial of Service (DoS) and can cause network resources to become unavailable. Arrival of IPv6 did not change basic principles of a flooding attack. However, with the introduction of new extension headers and ICMPv6 message types, along with integral multicast support, IPv6 may introduce more ways to develop malicious flooding attacks. In this case **Flood_Advertise6**, **Denial6**, **Flood_DHCPc6**, **Flood_MLD6**, **Flood_MLDrouter6**, **Flood_Solicitate6**, and **Fake_Router6** were chosen. For **Denial6** there are options for oversized Hop by Hop header and oversized destination header, for our purposes both were chosen.

Rogue Devices

A Rogue Device is any unauthorized device connected to the network that poses a significant risk to the organization. This is typically a router or server (e.g., DHCP) on a network which is not under the administrative control of the network administrator. It can be a device connected to the network by a user who may be either unaware of the consequences of their actions or may be knowingly using it for network attacks such as MITM or DoS. **Fake_Router6**, **Fake_MLD6**, and **Fake_MLDrouter6** (advertise) served this purpose in our tests.

Fragmentation

Fragmentation, covered in §2.4.2, deals with the misuse of fragment headers to perform DoS or avoid access controls such as Firewalls, Routers and NIDSs. IPv6 has eliminated fragmentation in the intermediate devices (e.g., Routers), however fragmentation can be performed at the end host. Malicious end hosts may attempt to avoid detection or conduct a DoS using fragmentation. For our test purposes we chose **Fragmentation6**, and the fragmentation options for **Fake_Router6**, and **Kill_Router6**.

Exploit

An exploit is a piece of software, a chunk of data, or sequence of commands that takes advantage of a vulnerability in an OS, application, or computer. Exploits generally cause unintended or unanticipated behavior to occur on the system being exploited. This frequently includes such things as gaining control of a computer system via privilege escalation or some type of DoS attack. To test this type of attack we transferred known malware via ftp, sent **Exploit6**, and used **Implementation6** from the THC toolkit.

Exploit6 performs attacks from various CVE [53] known IPv6 vulnerabilities on the target and reports back to the attacker upon completion. The Common Vulnerabilities and Exposure (CVE) is a dictionary of publicly known information security vulnerabilities and exposures that can be used for Intrusion Detection, vulnerability management and alerting, and patch management.

In the case of **Implementation6**, which looks for various OS implementation vulnerabilities, **Implementation6d** can also used as a listener on End_host to ensure appropriate results were passed. **Implementation6d** is generally used to ensure the attack can successfully operate through a firewall, which was not required for our tests since there was no firewall in our test bed.

3.3.1 The Tests

As stated in §3.3 most of our tests used attacks from the THC IPv6 toolkit [32]. The THC website has descriptions of these IPv6 attacks in a README file distributed with the toolkit. A broad range of attack types were chosen in an attempt to try to paint a good picture of overall IPv6 readiness of BRO and SNORT. In some cases, both IPv4 and IPv6 attacks were sent, however this was not true for every attack. For example, **Implementation6** is an IPv6 only attack, of which there is no equivalent in IPv4. Each NIDS used for this research was designed

in order to allow any rule or policy to detect attacks in both IPv4 and IPv6. In other words, if an attack has an IPv4 rule/policy then that same rule should detect the attack when sent in IPv6. To prove this we used our baseline attacks.

For testing purposes, each IPv6 attack was sent from Test_host to either of End_host or End_host2, while BRO_BSD, BRO2_server and SNORT_server, each loaded with their NIDS namesake passively monitored all network traffic. After each test was sent, manual log inspection was conducted at each NIDS to determine detection. For SNORT, detection results were logged in *Alerts.log*, and for BRO the detection results were logged in one of many log files, typically *Alarm.log* or *Weird.log*, found in */usr/local/bro/logs/current*. For BRO Version 2.0 (BRO2), detection results were also logged in one of many logs found in */usr/local/bro/logs/current*, typically *Weird.log* or *Notice.log*.

To test the deep packet inspection capabilities of BRO and SNORT, a piece of malware was sent from Test_host to End_host via File Transfer Protocol. The ftp server, running on End_host, utilized the VSFTPD [54] FTP server. The malware sent over ftp, named Slackbot, is an older piece of malware that was discovered on October 9, 2001 [55]. For detection purposes we wanted an older, well recognized piece of malware that would not be difficult to detect. To verify that it was easy to detect, we ran the copy we used in the test through a meta site called Jotti [56] that tests a particular file against 20 free online virus scanners. All 20 detected the file as malware making it a perfect selection for our tests.

When performing the actual transfer, Slackbot was sent three times. For two of the transfers, the malware was sent only as an executable (.exe) in order to determine if the NIDSs were inspecting the payload and would alert on this activity. The difference between the two initial transfers was that for the first transfer, the file was named *malware.exe*, but for the second transfer we removed the extension and renamed the file to *malware*. For the third transfer we compressed the *malware.exe* file into a zip file called *malware.zip* and then conducted the transfer. This was also done in order to determine how the NIDSs were inspecting packet payloads in an ftp transfer, as well as to find out if they would alert on any part of the payload name. This is discussed further in §4.3.

3.4 Transitional Test Case

To account for the period of time that our networks are neither completely IPv6 nor completely IPv4, a Transitional Test Case was created. This was done in order to cover the Transition,

Translation, and Tunneling attack class. Using the Transitional Test Bed displayed in Figure 3.2, each attack listed in the Attack Matrix shown in Table 3.1 was sent again from Test_host to End_host via End_host2 which was configured as a router (radvd). All log files were recreated in order to provide separate results for the Transitional Test Case.

3.5 Fuzz Testing

Fuzz testing as covered in §2.5, is generally a process that inputs invalid or unexpected random data to a piece of software or a computer. For our purposes, since we were only interested in testing the NIDS reaction to changes in the packets and streams, a slight variation from the traditional method was needed. Our intention was to fuzz the NIDSs and not a particular piece of software or host. To do this each NIDS was run inside GDB [57] while **Fuzz_ip6** [32] was run from Test_host against End_host. The options for **Fuzz_ip6** are listed in Table 3.2. Each test was conducted from Test_host and directed at End_host. Examples of the test formats are ***Fuzz_ip6*** *-IFSDHRJ -2 eth0 End_host* and ***Fuzz_ip6*** *-x -IFSDHRJ -2 eth0 End_host*. Once an exception or error was discovered, GDB preserved the state of the program execution stack. We then initiated a manual process of setting debugging break points and tracing the root cause of any traffic that caused the NIDS to crash.

Options:		
-1		fuzz ICMP6 echo request (default)
-2		fuzz ICMP6 neighbor solicitation
-3		fuzz ICMP6 neighbor advertisement
-4		fuzz ICMP6 router advertisement
-5		fuzz multicast listener report packet
-6		fuzz multicast listener done packet
-7		fuzz multicast listener query packet
-8		fuzz multicast listener v2 report packet
-9		fuzz multicast listener v2 query packet
-x		tries all 256 values for flag and byte types
-t	number	continue from test no. number
-T	number	only performs test no. number
-p	number	perform an alive check every number of tests (default: none)
-n	number	how many times to send each packet (default: 1)
-I		fuzz the IP + ICMP header too
-F		add one-shot fragmentation, and fuzz it too (for 1)
-S		add source-routing, and fuzz it too (for 1)
-D		add destination header, and fuzz it too (for 1)
-H		add hop-by-hop header, and fuzz it too (for 1 and 5-9)
-R		add router alert header, and fuzz it too (for 5-9 and all)
-J		add jumbo packet header, and fuzz it too (for 1)
		You can only define one of -1 ... -7, defaults to -1.

Table 3.2: Options for **Fuzz_ip6** From [32]

CHAPTER 4:
Detection Results

This chapter seeks to answer our primary research questions and thus determine the IPv6 readiness of the tested NIDSs. To do this, analysis of our detection data for each attack listed in Chapter 3 will be conducted in order to provide clear determination of detection.

4.1 "Native" Detection Results

The results for all tests sent in the "Native" environment are recorded in Table 4.2. A "Yes" result indicates that the attack resulted in a recognizable alert recorded in either BRO or SNORT's alert/alarm log. However, a "No" result means that either the NIDS did not have a rule or preprocessor written for that attack or that it was unable to correctly process the traffic. In SNORT's case a "No" means that an alert wasn't implemented in the default configuration or that a specific rule had not been, or in most cases could not be, included for that attack. We found that SNORT was able to initially detect 52 percent of the attacks sent. However, in all the tests conducted, the event log was constructed properly showing the correct type and amount of packets sent in the attack. Given the event log, it would not be difficult to write rules to alert on these events since SNORT is properly handling detection and logging of these events, in particular ICMPv6 type events.

4.1.1 BRO - BRO V2

In BRO, the IPv6 attacks and their detection results did not show the complete picture. By complete picture, we mean that the lack of detection did not represent what was actually happening inside the NIDS. In many of the test cases, multiple sent packets were not captured by BRO's event logging. For example, when conducting an intense IPv6 scan using NMAP [50] or *portscan6.py*, not all packets are analyzed. BRO alerted on these attacks and processed some but not all of the packets. Doing an NMAP one packet per port scan of all ports on End_host in an IPv4 configuration resulted in the alert shown in Figure 4.1. The alert for this port scan was accompanied with threshold messages at a certain number of ports scanned such as 50 or 250.

```
t=1320692142.221451 no=PortScan na=NOTICE_ALARM_ALWAYS es=bro sa=192.168.23.1
p=73/tcp num=50 msg=192.168.23.1\ has\ scanned\ 50\ ports\ of\ 192.168.23.2
tag=@1e-3b2d-1
t=1320692142.845071 no=PortScan na=NOTICE_ALARM_ALWAYS es=bro sa=192.168.23.1
p=276/tcp num=250 msg=192.168.23.1\ has\ scanned\ 250\ ports\ of\ 192.168.23.2
tag=@1e-3b2d-2
t=1320692145.374887 no=PortScan na=NOTICE_ALARM_ALWAYS es=bro sa=192.168.23.1
p=1027/tcp num=1000 msg=192.168.23.1\ has\ scanned\ 1000\ ports\ of\
192.168.23.2 tag=@1e-3b2d-3
```

Figure 4.1: IPv4 port scan results in BRO

In IPv6, when running the same NMAP scan (**NMAP6**), BRO gave the alert listed in Figure 4.2 called *LowPortTrolling*, this time without any threshold messages. These indications were also produced when running *portscan6.py*.

```
1323284282.244358 LowPortTrolling low port trolling 2001:6c0:1973::1 portmap
1323285193.950063 LowPortScanSummary 2001:6c0:1973::1 scanned a total of 18 low ports
1323285193.950063 PortScanSummary 2001:6c0:1973::1 scanned a total of 22 ports
```

Figure 4.2: IPv6 port scan results in BRO

This indicates that BRO was not processing all the events in IPv6 since the scans were traversing the same number of ports. Running Wireshark [58] in the background during the scans verified the same number of ports being scanned each time the attack was run. When terminating BRO, after performing these scans a small fraction of the total number of packets sent (46 out of 6000) were logged, which was another indication that BRO was having trouble creating event streams from IPv6 traffic. Similar behavior, where the IPv6 event log was not a complete picture, happened in every case where an attack was sent and not alerted on. BRO also had trouble with incomplete events or missing packets of the event in some attacks that were alerted on, such as in the Figure 4.2 example. We investigated these missing packets further by looking at BRO events in *events.bst* produced by adding the @load capture-events to the BRO configuration file in *local.bro*. During our investigation into this issue we noticed that the packet level events in the *event.bst*, as seen in Figure 4.3, were identical with the exception of the address field.

```
IPv4 port scan

Event [1322693709.773258] connection_rejected([id=[`=192.168.23.1, orig_p=20/tcp, resp_h=192.168.23.5, resp_p=1/tcp], orig=[size=0,
state=1], resp=[size=0, state=6], start_time=1322693709.77316, duration=9.89437103271484e-05, service={}, addl="", hot=0,
history="Sr"])

IPv6 port scan

Event [1322691439.100904] connection_rejected([id=[orig_h=2001:6c0:1973::1, orig_p=20/tcp, resp_h=2001:6c0:1973::5, resp_p=21/tcp],
orig=[size=0, state=1], resp=[size=0, state=6], start_time=1322691439.10061, duration=0.000295877456665039, service={}, addl="", hot=0,
history="Sr"])
```

Figure 4.3: Comparison of port scan event results in BRO

The results, displayed in Figures 4.3 and 4.4, show that BRO is capable of IPv6 detection when

the proper information is passed through its detection mechanisms. However in most of our attacks not all packets were being processed beyond the Event Engine level.

```
Event [1323380915.260397] connection_rejected([id=[orig_h=2001:6c0:1973::1, orig_p=20/tcp, resp_h=2001:6c0:1973::2, resp_p=110/tcp],
orig=[size=0, state=1], resp=[size=0, state=6], start_time=1323380915.2604, duration=1.9073486328125e-06, service={}, addl="", hot=0,
history="Sr"])Event [1323380915.262155] connection_rejected([id=[orig_h=2001:6c0:1973::1, orig_p=20/tcp, resp_h=2001:6c0:1973::2,
resp_p=111/tcp], orig=[size=0, state=1], resp=[size=0, state=6], start_time=1323380915.26215, duration=2.1457672119140 6e-06, service=
{}, addl="", hot=0, history="Sr"])

Event [1323380915.262155] notice_alarm([note=LowPortTrolling, msg="low port trolling 2001:6c0:1973::1 portmap", sub=<uninitialized>,
conn=<uninitialized>, iconn=<uninitialized>, id=<uninitialized>, src=2001:6c0:1973::1, dst=<uninitialized>, p=111/tcp,
user=<uninitialized>, filename=<uninitialized>, method=<uninitialized>, URL=<uninitialized>, n=<uninitialized>, aux=<uninitialized>,
action=NOTICE_ALARM_ALWAYS, src_peer=[id=0, host=127.0.0.1, p=0/unknown, is_local=T, descr="", class=<uninitialized>], tag="1d-7142-2",
dropped=<uninitialized>, captured=<uninitialized>, do_alarm=<uninitialized>]NOTICE_ALARM_ALWAYS)

Event [1323380915.262155] notice_action([note=LowPortTrolling, msg="low port trolling 2001:6c0:1973::1 portmap", sub=<uninitialized>,
conn=<uninitialized>, iconn=<uninitialized>, id=<uninitialized>, src=2001:6c0:1973::1, dst=<uninitialized>, p=111/tcp,
user=<uninitialized>, filename=<uninitialized>, method=<uninitialized>, URL=<uninitialized>, n=<uninitialized>, aux=<uninitialized>,
action=NOTICE_ALARM_ALWAYS, src_peer=[id=0, host=127.0.0.1, p=0/unknown, is_local=T, descr="", class=<uninitialized>], tag="1d-7142-2",
dropped=<uninitialized>, captured=<uninitialized>, do_alarm=<uninitialized>]NOTICE_ALARM_ALWAYS)

Event [1323380915.275078] connection_rejected([id=[orig_h=2001:6c0:1973::1, orig_p=20/tcp, resp_h=2001:6c0:1973::2, resp_p=113/tcp],
orig=[size=0, state=1], resp=[size=0, state=6], start_time=1323380915.27498, duration=0.0001001358032226 56, service={}, addl="", hot=0,
history="Sr"])
```

Figure 4.4: BRO *Events.bst* log for IPv6 port scan

Even though BRO has a policy in place and is receiving the appropriate packets at the Packet Capture level, it fails to detect IPv6 attacks that it is capable of detecting in IPv4. This indicates that BRO may not be reconstructing event streams at the Event Engine level or that software incompatibilities exist. In an attempt to find these software incompatibilities, we ran BRO inside GDB [57] for debugging purposes. The results of our debugging, however, revealed no one clear problem with BRO software and remains an item for future work.

When running **NMAP6** and *portscan6.py* in BRO V2 (BRO2) we noticed that most (consistently more that 90 percent) if not all packets were being dropped and logged in the *Notice.log*. We considered this as an indication that BRO2 used a policy in order to determine that dropping these packets was required, although since every TCP connection request and rejection were logged in the *conn.log*, meaning that BRO2 had indications of a vertical port scan and did not alert on it, we did not consider this a detection of the attack. Similar activity was seen in a few other attacks, such as **SMURF6** and **Denial6**, and again not considered as detection.

Detect-New-IPv6

In BRO2, after running **Detect-new-IPv6**, we discovered ”bad UDP checksum” activity in the *Weird.log*. After looking at the packets, we determined that the checksums were actually correct at both End_host and at the NIDS, therefor we did not consider this activity as detection.

Flood_DHCPv6

In other cases, we also considered similar activity in BRO’s *Weird.log* as a sign of non-detection. For example, in the case of **Flood_DHCPc6** we considered the “Bad UDP checksum” message in *Weird.log* to be a sign of the NIDS mis-interpreting the packets. Figure 4.5 shows the output

from BRO's *Weird.log* for the attack. BRO2 showed the exact same activity in its *Weird.log* for this attack.

```
1323973233.013373 fe80::100:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1323973233.013395 fe80::200:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1323973233.013397 fe80::300:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1323973233.013630 fe80::400:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1323973233.013634 fe80::500:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1323973233.013733 fe80::600:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1323973233.013852 fe80::700:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1323973233.013856 fe80::800:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1323973233.013980 fe80::900:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1323973233.013984 fe80::a00:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1323973233.014112 fe80::b00:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1323973233.014116 fe80::c00:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1323973233.014213 fe80::d00:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1323973233.014338 fe80::e00:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1323973233.014342 fe80::f00:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1323973233.014464 fe80::1000:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1323973233.014468 fe80::1100:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1323973233.014591 fe80::1200:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1323973233.014595 fe80::1300:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1323973233.014716 fe80::1400:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1323973233.014720 fe80::1500:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1323973233.015200 fe80::1600:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1323973233.015204 fe80::1700:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1323973233.015206 fe80::1800:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1323973233.015208 fe80::1900:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1323973233.015210 fe80::1a00:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
```

Figure 4.5: Entry for **Flood_DHCPc6** in *Weird.log* for BRO

4.1.2 SNORT

SNORT's performance, as discussed in §4.1, indicated a higher level of IPv6 readiness. This subsection presents detailed detection results.

Alive6

SNORT detected **Alive6**, shown in Figure 4.6, as "ICMP type not decoded" and lists the link local addresses for both the source and destination.

```
[**] [116:431:1] (snort_decoder) WARNING: ICMP6 type not decoded [**]
[Classification: Misc activity] [Priority: 3]
12/21-09:46:56.772410 fe80:0000:0000:0000:0a00:27ff:feb3:9fb3 ->
fe80:0000:0000:
0000:0a00:27ff:fe35:9857
IPV6-ICMP TTL:255 TOS:0x0 ID:0 IpLen:40 DgmLen:72

[**] [116:431:1] (snort_decoder) WARNING: ICMP6 type not decoded [**]
[Classification: Misc activity] [Priority: 3]
12/21-09:46:56.772432 fe80:0000:0000:0000:0a00:27ff:fe35:9857 ->
fe80:0000:0000:
0000:0a00:27ff:feb3:9fb3
IPV6-ICMP TTL:255 TOS:0x0 ID:0 IpLen:40 DgmLen:64
```

Figure 4.6: SNORT output for **Alive6**

Figure 4.7 displays the detection results for **Alive6** with the *-S 2* invalid header option.

```
[**] [116:281:1] (snort_decoder) WARNING: IPv6 header includes an invalid
value for the "next header" field [**]
[Classification: Generic Protocol Command Decode] [Priority: 3]
12/21-10:04:04.063235 2001:06c0:1973:0000:0000:0000:0000:0001 ->
ff02:0000:0000:0000:0000:0000:0000:0001
PROTO:128 TTL:255 TOS:0x0 ID:0 IpLen:40 DgmLen:72
```

Figure 4.7: SNORT output for **Alive6** with *-S 2* option

Figure 4.8 displays the detection results for **Alive6** with the *-S 4* invalid hop-by-hop header option. This is an IPv4 rule event that is generated when a possibly crafted ICMP Source Quench datagram (ICMP Type 4) is detected without the corresponding code specification, which is a sign of malicious activity.

```
[**] [1:18474:1] ICMPv6 Echo Request [**]
[Classification: Misc activity] [Priority: 3]
11/02-11:02:36.826072 2001:06c0:1973:0000:0000:0000:0000:0001 -> ff02:0000:0000:0000:0000:0000:0000:0001
IPV6-ICMP TTL:255 TOS:0x0 ID:3372665094 IpLen:40 DgmLen:72 RB

[**] [1:448:7] ICMP Source Quench undefined code [**]
[Classification: Misc activity] [Priority: 3]
11/02-11:02:36.826086 2001:06c0:1973:0000:0000:0000:0000:0002 -> 2001:06c0:1973:0000:0000:0000:0000:0001
IPV6-ICMP TTL:64 TOS:0x0 ID:0 IpLen:40 DgmLen:120

[**] [1:448:7] ICMP Source Quench undefined code [**]
[Classification: Misc activity] [Priority: 3]
11/02-11:02:36.826087 2001:06c0:1973:0000:0000:0000:0000:0002 -> 2001:06c0:1973:0000:0000:0000:0000:0001
IPV6-ICMP TTL:64 TOS:0x0 ID:0 IpLen:40 DgmLen:120
```

Figure 4.8: SNORT output for **Alive6** with *-S 4* option

Portscan6.py

The results of *portscan6.py* are listed in Figure 4.9. Notice that SNORT detected "Bad-Traffic TCP port 0", this is because *portscan6.py* starts scanning at port 0 and is a valid detection.

```
[**] [116:446:1] (snort_decoder) WARNING: BAD-TRAFFIC TCP port 0 traffic [**]
[Classification: Misc activity] [Priority: 3]
11/25-16:16:25.564163 2001:06c0:1973:0000:0000:0000:0000:0001:20 ->
2001:06c0:1973:0000:0000:0000:0000:0005:0
TCP TTL:64 TOS:0x0 ID:0 IpLen:40 DgmLen:60
******S* Seq: 0x0  Ack: 0x0  Win: 0x2000  TcpLen: 20

[**] [116:446:1] (snort_decoder) WARNING: BAD-TRAFFIC TCP port 0 traffic [**]
[Classification: Misc activity] [Priority: 3]
11/25-16:16:25.564283 2001:06c0:1973:0000:0000:0000:0000:0005:0 ->
2001:06c0:1973:0000:0000:0000:0000:0001:20
TCP TTL:64 TOS:0x0 ID:0 IpLen:40 DgmLen:60
***A*R** Seq: 0x0  Ack: 0x1  Win: 0x0  TcpLen: 20

[**] [116:431:1] (snort_decoder) WARNING: ICMP6 type not decoded [**]
[Classification: Misc activity] [Priority: 3]
11/25-16:16:30.560400 fe80:0000:0000:0000:0a00:27ff:fe03:face ->
2001:06c0:1973:0000:0000:0000:0000:0001
IPV6-ICMP TTL:255 TOS:0x0 ID:0 IpLen:40 DgmLen:72
```

Figure 4.9: SNORT alert from *portscan6.py*

NMAP6

Figure 4.10 illustrates the SNORT detection results from **NMAP6**.

```
[**] [122:1:1] PSNG_TCP_PORTSCAN [**]
[Classification: Attempted Information Leak] [Priority: 2]
11/25-16:24:45.528430 2001:06c0:1973:0000:0000:0000:0000:0001 ->
2001:06c0:1973:0000:0000:0000:0000:0005
PROTO:255 TTL:64 TOS:0x0 ID:0 IpLen:40 DgmLen:236

[**] [116:431:1] (snort_decoder) WARNING: ICMP6 type not decoded [**]
[Classification: Misc activity] [Priority: 3]
11/25-16:24:50.532351 fe80:0000:0000:0000:0a00:27ff:fe03:face ->
2001:06c0:1973:0000:0000:0000:0000:0001
IPV6-ICMP TTL:255 TOS:0x0 ID:0 IpLen:40 DgmLen:72
```

Figure 4.10: SNORT alert from **NMAP6**

Toobig6

The detection results from **Toobig6** are shown in Figure 4.11. The detection of the *"ICMPv6 packet of type 2 (message too big)"* including *ICMP unassigned type 2*, is an exact detection.

```
[**] [116:285:1] (snort_decoder) WARNING: ICMPv6 packet of type 2 (message too big) with MTU field < 1280 [**]
[Classification: Generic Protocol Command Decode] [Priority: 3]
03/09-14:48:49.803056 2001:06c0:1972:0000:0000:0000:0000:0001 -> 2001:06c0:1973:0000:0000:0000:0000:0002
IPV6-ICMP TTL:255 TOS:0x0 ID:0 IpLen:40 DgmLen:549

[**] [1:1394:12] SHELLCODE x86 inc ecx NOOP [**]
[Classification: Executable Code was Detected] [Priority: 1]
03/09-14:48:49.803056 2001:06c0:1972:0000:0000:0000:0000:0001 -> 2001:06c0:1973:0000:0000:0000:0000:0002
IPV6-ICMP TTL:255 TOS:0x0 ID:0 IpLen:40 DgmLen:549

[**] [1:460:8] ICMP unassigned type 2 [**]
[Classification: Misc activity] [Priority: 3]
03/09-14:48:49.803056 2001:06c0:1972:0000:0000:0000:0000:0001 -> 2001:06c0:1973:0000:0000:0000:0000:0002
IPV6-ICMP TTL:255 TOS:0x0 ID:0 IpLen:40 DgmLen:549
```

Figure 4.11: SNORT detection for **Toobig6**

Fake_DHCPS6

Figure 4.12 displays the SNORT output from **Fake_DHCPS6**, showing Test_host advertising itself as a DHCP server to the all_hosts multicast address (FF02::1) and the subsequent invalid and ICMPv6 traffic.

```
[**] [116:431:1] (snort_decoder) WARNING: ICMP6 type not decoded [**]
[Classification: Misc activity] [Priority: 3]
12/06-11:01:42.512628 2001:06c0:1973:0000:0000:0000:0000:0001 -> ff02:0000:0000:0000:00
00:0000:0000:0001
IPV6-ICMP TTL:255 TOS:0x0 ID:0 IpLen:40 DgmLen:72

[**] [116:431:1] (snort_decoder) WARNING: ICMP6 type not decoded [**]
[Classification: Misc activity] [Priority: 3]
12/06-11:01:42.512630 2001:06c0:1973:0000:0000:0000:0000:0002 -> 2001:06c0:1973:0000:0000:0000:0000:0001
IPV6-ICMP TTL:255 TOS:0x0 ID:0 IpLen:40 DgmLen:72

[**] [116:275:1] (snort_decoder) WARNING: IP dgm len > captured len! [**]
12/06-11:01:42.512875

[**] [116:275:1] (snort_decoder) WARNING: IP dgm len > captured len! [**]
12/06-11:01:42.512876
```

Figure 4.12: SNORT detection for **Fake_DHCPS6**

RSMURF6

SNORT detection results for **RSMURF6** are shown in Figure 4.13. This attack resulted in several different SNORT alerts, each of which could be used to write a rule for its detection in IPv6. For instance, this attacks solicits a reply from the target (End_host) which is an ICMPv6 type 129 (echo reply) to the all-hosts multicast address (FF02::1). That traffic alerts SNORT and the resulting “ICMP6 packet to multicast address” warning is issued.

```
[**] [116:432:1] (snort_decoder) WARNING: ICMP6 packet to multicast address [**]
[Classification: Misc activity] [Priority: 3]
12/19-12:20:52.145981 fe80:0000:0000:0000:0a00:27ff:fe35:9857 -> ff02:0000:0000:
0000:0000:0000:0000:0001
IPV6-ICMP TTL:1 TOS:0x0 ID:0 IpLen:40 DgmLen:64

[**] [1:18473:1] ICMPv6 Echo Reply [**]
[Classification: Misc activity] [Priority: 3]
12/19-12:20:52.145981 fe80:0000:0000:0000:0a00:27ff:fe35:9857 -> ff02:0000:0000:
0000:0000:0000:0000:0001
IPV6-ICMP TTL:1 TOS:0x0 ID:0 IpLen:40 DgmLen:64

[**] [116:151:1] (snort decoder) WARNING: Bad Traffic Same Src/Dst IP [**]
[Classification: Potentially Bad Traffic] [Priority: 2]
12/19-12:20:52.145983 ff02:0000:0000:0000:0000:0000:0000:0001 -> ff02:0000:0000:
0000:0000:0000:0000:0001
IPV6-ICMP TTL:255 TOS:0x0 ID:0 IpLen:40 DgmLen:64
[Xref => http://www.microsoft.com/technet/security/bulletin/ms05-019.mspx][Xref
=> http://www.securityfocus.com/bid/2666][Xref => http://cve.mitre.org/cgi-bin/c
vename.cgi?name=2005-0688][Xref => http://cve.mitre.org/cgi-bin/cvename.cgi?name
=1999-0016]

[**] [116:432:1] (snort_decoder) WARNING: ICMP6 packet to multicast address [**]
[Classification: Misc activity] [Priority: 3]
12/19-12:20:52.145986 fe80:0000:0000:0000:0a00:27ff:fe0a:e426 -> ff02:0000:0000:0000:0000:0000:0000:0001
IPV6-ICMP TTL:1 TOS:0x0 ID:0 IpLen:40 DgmLen:64

[**] [1:18473:1] ICMPv6 Echo Reply [**]
[Classification: Misc activity] [Priority: 3]
12/19-12:20:52.145986 fe80:0000:0000:0000:0a00:27ff:fe0a:e426 -> ff02:0000:0000:0000:0000:0000:0000:0001
IPV6-ICMP TTL:1 TOS:0x0 ID:0 IpLen:40 DgmLen:64

[**] [116:432:1] (snort_decoder) WARNING: ICMP6 packet to multicast address [**]
[Classification: Misc activity] [Priority: 3]
12/19-12:20:52.145999 fe80:0000:0000:0000:0a00:27ff:fe35:9857 -> ff02:0000:0000:0000:0000:0000:0000:0001
IPV6-ICMP TTL:1 TOS:0x0 ID:0 IpLen:40 DgmLen:64
```

Figure 4.13: SNORT detection for **RSMURF6**

Flood_Advertise6

Figure 4.14 displays the detection results for **Flood_Advertise6**, where the three alerts listed appear to be from different IP addresses going to the all_host multicast address. With closer inspection, however, each of these three addresses share the same first 80 bits (`fe80:0000:0000:0000:0218`) which could be an indication of one host generating random addresses to flood advertisements on the network segment.

```
[**] [116:431:1] (snort_decoder) WARNING: ICMP6 type not decoded [**]
[Classification: Misc activity] [Priority: 3]
03/07-13:31:22.063386 fe80:0000:0000:0000:0218:58ff:fe0d:ca71 ->
ff02:0000:0000:0000:0000:0000:0000:0001
IPV6-ICMP TTL:255 TOS:0x0 ID:0 IpLen:40 DgmLen:72

[**] [116:431:1] (snort_decoder) WARNING: ICMP6 type not decoded [**]
[Classification: Misc activity] [Priority: 3]
03/07-13:31:22.063388 fe80:0000:0000:0000:0218:f7ff:febe:e616 ->
ff02:0000:0000:0000:0000:0000:0000:0001
IPV6-ICMP TTL:255 TOS:0x0 ID:0 IpLen:40 DgmLen:72

[**] [116:431:1] (snort_decoder) WARNING: ICMP6 type not decoded [**]
[Classification: Misc activity] [Priority: 3]
03/07-13:31:22.063389 fe80:0000:0000:0000:0218:e0ff:fe58:8642 ->
ff02:0000:0000:0000:0000:0000:0000:0001
IPV6-ICMP TTL:255 TOS:0x0 ID:0 IpLen:40 DgmLen:72
```

Figure 4.14: SNORT results for **Flood_Advertise6**

Denial6

Figure 4.15 illustrates the detection results from **Denial6** test case 1 (large hop-by-hop headers with router-alert filled with unknown options). It would be difficult, but not impossible, to write a SNORT rule for this traffic. Since there is a large amount of traffic between the two hosts, and each packet contains options with the same value, ID:516 for example, there are pieces that could be used to build a detection rule. On the other hand, **Denial6** using test case 2 (large destination header filed with unknown options) resulted in an output of only ICMP echo requests and replies from the two hosts and would require more work in order to develop a rule.

```
[**] [1:18474:1] ICMPv6 Echo Request [**]
[Classification: Misc activity] [Priority: 3]
11/18-08:42:25.559638 2001:06c0:1973:0000:0000:0000:0000:0001 ->
2001:06c0:1973:0000:0000:0000:0000:0002
IPV6-ICMP TTL:255 TOS:0x0 ID:516 IpLen:40 DgmLen:1480

[**] [1:18474:1] ICMPv6 Echo Request [**]
[Classification: Misc activity] [Priority: 3]
11/18-08:42:25.559654 2001:06c0:1973:0000:0000:0000:0000:0001 ->
2001:06c0:1973:0000:0000:0000:0000:0002
IPV6-ICMP TTL:255 TOS:0x0 ID:516 IpLen:40 DgmLen:1480

[**] [1:18474:1] ICMPv6 Echo Request [**]
[Classification: Misc activity] [Priority: 3]
11/18-08:42:25.559656 2001:06c0:1973:0000:0000:0000:0000:0001 ->
2001:06c0:1973:0000:0000:0000:0000:0002
IPV6-ICMP TTL:255 TOS:0x0 ID:516 IpLen:40 DgmLen:1480

[**] [1:18474:1] ICMPv6 Echo Request [**]
[Classification: Misc activity] [Priority: 3]
11/18-08:42:25.559657 2001:06c0:1973:0000:0000:0000:0000:0001 ->
2001:06c0:1973:0000:0000:0000:0000:0002
IPV6-ICMP TTL:255 TOS:0x0 ID:516 IpLen:40 DgmLen:1480

[**] [1:18474:1] ICMPv6 Echo Request [**]
[Classification: Misc activity] [Priority: 3]
11/18-08:42:25.559658 2001:06c0:1973:0000:0000:0000:0000:0001 ->
2001:06c0:1973:0000:0000:0000:0000:0002
IPV6-ICMP TTL:255 TOS:0x0 ID:516 IpLen:40 DgmLen:1480
```

Figure 4.15: SNORT detection for **Denial6** with large hop-by-hop headers

Fragmentation6

Fragmentation6 resulted in multiple SNORT alerts listed in Figure 4.16. We again see `SHELLCODE x86 inc ecx NOOP` which is an indicator rule that is meant to be used in conjunction with other rules for a more complete picture, along with several fragmentation alerts.

```
[**] [1:1394:12] SHELLCODE x86 inc ecx NOOP [**]
[Classification: Executable Code was Detected] [Priority: 1]
12/08-17:39:57.663776 2001:06c0:1973:0000:0000:0000:0000:0002 -> 2001:06c0:1973:0000:0000:0000:0000:0001
IPV6-ICMP TTL:64 TOS:0x0 ID:0 IpLen:40 DgmLen:271

[**] [123:10:1] (spp_frag3) Bogus fragmentation packet. Possible BSD attack [**]
[Classification: Attempted Administrator Privilege Gain] [Priority: 1]
12/08-17:39:59.675953 2001:06c0:1973:0000:0000:0000:0000:0001 -> 2001:06c0:1973:0000:0000:0000:0000:0002
IPV6-FRAG TTL:64 TOS:0x7 ID:3735881804 IpLen:40 DgmLen:48
Frag Offset: 0x0000   Frag Size: 0x0008

[**] [123:5:1] (spp_frag3) Zero-byte fragment packet [**]
[Classification: Attempted Denial of Service] [Priority: 2]
12/08-17:39:59.675953 2001:06c0:1973:0000:0000:0000:0000:0001 -> 2001:06c0:1973:0000:0000:0000:0000:0002
IPV6-FRAG TTL:64 TOS:0x7 ID:3735881804 IpLen:40 DgmLen:48
Frag Offset: 0x0000   Frag Size: 0x0008

[**] [123:10:1] (spp_frag3) Bogus fragmentation packet. Possible BSD attack [**]
[Classification: Attempted Administrator Privilege Gain] [Priority: 1]
12/08-17:40:00.683657 2001:06c0:1973:0000:0000:0000:0000:0001 -> 2001:06c0:1973:0000:0000:0000:0000:0002
IPV6-FRAG TTL:64 TOS:0x8 ID:3735881805 IpLen:40 DgmLen:49
Frag Offset: 0x0000   Frag Size: 0x0009

[**] [123:10:1] (spp_frag3) Bogus fragmentation packet. Possible BSD attack [**]
[Classification: Attempted Administrator Privilege Gain] [Priority: 1]
12/08-17:40:03.679806 2001:06c0:1973:0000:0000:0000:0000:0001 -> 2001:06c0:1973:0000:0000:0000:0000:0002
IPV6-FRAG TTL:64 TOS:0x0 ID:3735881806 IpLen:40 DgmLen:48
Frag Offset: 0x0000   Frag Size: 0x0008

[**] [123:5:1] (spp_frag3) Zero-byte fragment packet [**]
[Classification: Attempted Denial of Service] [Priority: 2]
12/08-17:40:03.679806 2001:06c0:1973:0000:0000:0000:0000:0001 -> 2001:06c0:1973:0000:0000:0000:0000:0002
IPV6-FRAG TTL:64 TOS:0x0 ID:3735881806 IpLen:40 DgmLen:48
Frag Offset: 0x0000   Frag Size: 0x0008

[**] [123:10:1] (spp_frag3) Bogus fragmentation packet. Possible BSD attack [**]
[Classification: Attempted Administrator Privilege Gain] [Priority: 1]
12/08-17:40:06.669628 2001:06c0:1973:0000:0000:0000:0000:0001 -> 2001:06c0:1973:0000:0000:0000:0000:0002
IPV6-FRAG TTL:64 TOS:0x0 ID:3735881807 IpLen:40 DgmLen:49
Frag Offset: 0x0000   Frag Size: 0x0009

[**] [123:2:1] (spp_frag3) Teardrop attack [**]
[Classification: Attempted Denial of Service] [Priority: 2]
12/08-17:40:06.669628 2001:06c0:1973:0000:0000:0000:0000:0001 -> 2001:06c0:1973:0000:0000:0000:0000:0002
IPV6-FRAG TTL:64 TOS:0x0 ID:3735881807 IpLen:40 DgmLen:49
Frag Offset: 0x0000   Frag Size: 0x0009
```

Figure 4.16: SNORT detection for **Fragmentation6**

Exploit6

The SNORT output for **Exploit6** are not so clearly detectable, this due to the fact that this attack has multiple test cases based on multiple known CVE listed attacks. As seen in Figure 4.17, the attacker (Test_host) sends an ICMPv6 packet to End_host which is immediately followed by invalid packets.

```
[**] [116:431:1] (snort_decoder) WARNING: ICMP6 type not decoded [**]
[Classification: Misc activity] [Priority: 3]
12/06-11:01:42.512628 2001:06c0:1973:0000:0000:0000:0000:0001 -> ff02:0000:0000:0000:00
00:0000:0000:0001
IPV6-ICMP TTL:255 TOS:0x0 ID:0 IpLen:40 DgmLen:72

[**] [116:431:1] (snort_decoder) WARNING: ICMP6 type not decoded [**]
[Classification: Misc activity] [Priority: 3]
12/06-11:01:42.512630 2001:06c0:1973:0000:0000:0000:0000:0002 -> 2001:06c0:1973:0000:0000:0000:0000:0001
IPV6-ICMP TTL:255 TOS:0x0 ID:0 IpLen:40 DgmLen:72

[**] [116:275:1] (snort_decoder) WARNING: IP dgm len > captured len! [**]
12/06-11:01:42.512875

[**] [116:275:1] (snort_decoder) WARNING: IP dgm len > captured len! [**]
12/06-11:01:42.512876

|
```

Figure 4.17: SNORT output for **Exploit6**

Other "Native" Detection Results

Table 4.1 lists all attacks that presented "ICMP type not decoded" as an alert in SNORT's *Alert.log*. This alert is an indication that the IPv6 packets from the attacks are getting processed correctly up to SNORT's Detection Engine (see Figure §2.13), denoting that the only thing preventing exact detection is a specific rule, which can be tailored from inspections of the attack's traffic, for each particular attack.

Fake_Advertise6	Fake_DNS6	DoS-new-IPv6
Detect-new-IPv6	NDPexhaust	SMURF6
Fake_MLDrouter6	Fake_MLD26	Kill_router6
Parasite6	Flood_DHCPc6	Flood_MLD6
Flood_MLDrouter6	Flood_Solicitate6	Fake_MLD6
Fake_Router6	Implementation6	

Table 4.1: List of SNORT "ICMP type not decoded" detections

Table 4.2 displays the results of all "Native" tests.

"Native" Detection Matrix

			Detection		
Attack Type			BRO	BRO2	SNORT
	IPv4 Attack	**IPv6 Attack**			
Reconnaissance					
	Port Scan	Alive6(resolve)	No	No	Yes
	Port Scan	Alive6(Inv. Hdr)	No	No	Yes
	Port Scan	Alive6(Inv. Hop by Hop)	No	No	Yes
	portscan4.py	portscan6.py	Yes**	No+	Yes*
	NMAP	NMAP6(scan ports)	Yes**	No+	Yes
	NMAP	NMAP6(TCP scan)	Yes**	No	Yes
Unauthorized Access					
	ARP Poisoning(ICMP)	Fake_Advertise6	No	No	Yes
	DNS	Fake_DNS6	No	No	Yes
	Route Implanting	Toobig6	No	No	Yes
Host Configuration					
	DHCP DoS	DoS-new-IPv6	No	No	Yes
		Detect-new-IPv6	No	No+	Yes
		Fake_DHCPS6	No	No	Yes
		NDPexhaust	No	No+	Yes
Broadcast Amplification Attacks(SMURF)					
	SMURF	SMURF6	No	No+	Yes
		RSMURF6	No	No+	Yes
Routing Attacks					
	Various DoS	Fake_Router6	No	No	No
		Fake_MLDrouter6(Sol.)	No	No	Yes
		Fake_MLDrouter6(Term.)	No	No	Yes
		Fake_MLD26(Query)	No	No	Yes
		Fake_MLD26(Add)	No	No	Yes
		Fake_MLD26(Delete)	No	No	Yes
		Kill_router6	No	No	Yes

"Native" Detection Matrix Cont.

			Detection		
Attack Type			BRO	BRO2	SNORT
	IPv4 Attack	**IPv6 Attack**			
Man in the Middle (MITM)					
	Various	Parasite6	No	No	Yes
Flooding					
	Various DoS	Denial6(lrg Hop by Hop)	No	No	Yes
		Denial6(large dst Hdr)	No	No+	Yes
	DHCP flooding	Flood_DHCPc6	No+	No+	Yes
	Various	Flood_Advertise6	No	No	Yes
		Flood_MLD6	No	No	Yes
		Flood_MLDrouter6	No	No	Yes
		Flood_Solicitate6(Network)	No	No	Yes
		Flood_Solicitate6(Target IP)	No	No	Yes
Rogue Devices					
	Rogue Router	Fake_Router6	No	No	No
		Fake_MLD6	No	No	Yes
		Fake_MLDrouter6(Adv.)	No	No	Yes
Fragmentation					
	Fragmentation	Fragmentation6	No	No	Yes
		Fake_Router6(Frag)	No	No	Yes
		Kill_Router6(Frag)	No	No	Yes
Exploit					
	Malware Transfer	FTP Malware	No	No	Yes#
	Various	Exploit6	No	No+	Yes
		Implementation6	No	Yes(w)	Yes

Table 4.2: Detection Matrix —— Notes:* our Scapy portscan6 scanned TCP port 0 which was detected by SNORT, ** BRO detected after we enabled event tracing, # Read string "Malware.exe", + dropped packet activity in *Notice.log*, (w) *Weird.log* activity.

4.2 Transitional Detection Results

The results for all tests sent in the Transitional environment are recorded in Table 4.3. Just as in §4.1, a "Yes" result indicates that the attack resulted in a recognizable alert being recorded in either BRO or SNORT's alert/alarm log and a "No" result meant that an alert was not issued for that attack. For SNORT, just as illustrated in §4.1, all events were constructed and logged appropriately, which meant that detection was simply a matter of writing rules rather than modification of the NIDS itself. As covered in §3.1.1, the period of time in which networks are operating with both IPv4 and IPv6 addresses is considered Transitional. We do not expect NIDS performance to improve during this period since the NIDS will be required to detect both IPv4 and IPv6 attacks, while at the same time be able to detect attacks from packets inside a tunneled environment.

4.2.1 BRO - BRO V2

As in §4.1.1, BRO's performance in the Transitional environment fell short of readiness. Again, we discovered that BRO and BRO2 were not processing all events and that our detection results did not convey any indication of true readiness or IPv6 operability of the NIDS.

Portscan6.py

Figure 4.18 shows the simple alert log entry in BRO for *portscan6.py*. Just as in the "Native" environment, IPv6 port scan events were not processed completely.

```
1324419882.041558 LowPortTrolling low port trolling 2001:6c0:1973::1 portmap
```

Figure 4.18: IPv6 Transitional port scan results in BRO

NMAP6

The Transitional results for **NMAP6**, shown in Figure 4.19, are just as simple as with *portscan6.py* and again not a representation of the entire event or of all packets sent. BRO2, just as in §4.1.1, showed all connection request/rejections in the *conn.log* and all packets being dropped in *Weird.log* for *portscan6.py* and **NMAP6**.

```
1324491329.607859 LowPortTrolling low port trolling 2001:6c0:1973::1 587/tcp
(END)
```

Figure 4.19: BRO Transitional NMAP6 results

Detect-New-IPv6

For **Detect-new-IPv6** in the Transitional environment we again saw activity in BRO's *Weird.log*, shown in Figure 4.20, which we did not consider detection.

```
1324493975.873760 2001:6c0:1973::5/5353 > ff02::fb/5353: bad_UDP_checksum
1324493976.573333 2001:6c0:1973::5/5353 > ff02::fb/5353: bad_UDP_checksum
1324493977.951284 2001:6c0:1973::5/5353 > ff02::fb/5353: bad_UDP_checksum
1324493978.652856 2001:6c0:1973::5/5353 > ff02::fb/5353: bad_UDP_checksum
1324497635.636091 fe80::a00:27ff:fe03:face/5353 > ff02::fb/5353: bad_UDP_checksum
1324497635.636127 fe80::a00:27ff:fe0b:37c/5353 > ff02::fb/5353: bad_UDP_checksum
1324497635.685112 fe80::a00:27ff:fe03:face/5353 > ff02::fb/5353: bad_UDP_checksum
1324497635.885913 fe80::a00:27ff:fe03:face/5353 > ff02::fb/5353: bad_UDP_checksum
1324497636.137474 fe80::a00:27ff:fe03:face/5353 > ff02::fb/5353: bad_UDP_checksum
1324497636.137518 fe80::a00:27ff:fe0b:37c/5353 > ff02::fb/5353: bad_UDP_checksum
1324497636.337762 fe80::a00:27ff:fe03:face/5353 > ff02::fb/5353: bad_UDP_checksum
1324497636.884960 fe80::a00:27ff:fe03:face/5353 > ff02::fb/5353: bad_UDP_checksum
1324497637.576303 fe80::a00:27ff:fe03:face/5353 > ff02::fb/5353: bad_UDP_checksum
```

Figure 4.20: *Weird.log* results for **Detect-new-IPv6** in Transitional BRO

For BRO2, just as in §4.1.1, we discovered "bad UDP checksum" activity in the *Weird.log* and again we did not consider this detection.

Flood_DHCPc6

In the Transitional environment BRO and BRO2 interpreted **Flood_DHCPc6** as "Bad UDP checksum" messages in the *Weird.log*, just as in §4.1.1 and displayed in Figure 4.21. This was also not considered as detection.

```
1325184977.470774 fe80::100:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1325184977.471256 fe80::200:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1325184977.471262 fe80::300:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1325184977.471267 fe80::400:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1325184977.471273 fe80::500:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1325184977.471278 fe80::600:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1325184977.471284 fe80::700:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1325184977.471289 fe80::800:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1325184977.471378 fe80::900:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1325184977.471380 fe80::a00:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1325184977.471380 fe80::b00:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1325184977.471453 fe80::c00:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1325184977.471533 fe80::d00:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1325184977.471534 fe80::e00:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1325184977.471631 fe80::f00:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1325184977.471634 fe80::1000:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1325184977.472191 fe80::1100:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1325184977.472193 fe80::1200:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1325184977.472194 fe80::1300:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1325184977.472194 fe80::1400:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1325184977.472195 fe80::1500:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1325184977.472196 fe80::1600:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1325184977.472197 fe80::1700:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1325184977.472272 fe80::1800:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1325184977.472355 fe80::1900:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1325184977.472356 fe80::1a00:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1325184977.472428 fe80::1b00:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1325184977.478195 fe80::1c00:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1325184977.478276 fe80::1d00:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1325184977.478359 fe80::1e00:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1325184977.478361 fe80::1f00:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1325184977.478432 fe80::2000:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1325184977.478504 fe80::2100:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1325184977.478575 fe80::2200:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1325184977.478662 fe80::2300:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1325184977.478663 fe80::2400:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1325184977.478735 fe80::2500:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1325184977.479179 fe80::2600:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1325184977.479180 fe80::2700:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
1325184977.479181 fe80::2800:0:0:0/546 > ff02::1:2/547: bad_UDP_checksum
```

Figure 4.21: BRO Transitional results for **Flood_DHCPc6**

4.2.2 SNORT

As discussed in §4.2 we expected SNORT, in the Transitional scenario, to performed similarly well as compared to the "Native" scenario (§4.1.2). As expected, the detection of attacks in the Transitional environment paralleled SNORT's results in the "Native" environment. This section will illustrate some of the SNORT Transitional detection results.

Portscan6.py

Figure 4.22 shows the results from *portscan6.py*. Unlike what is shown in §4.9, SNORT's detection of *portscan6.py* included an alert for a TCP port scan.

```
[**] [116:446:1] (snort_decoder) WARNING: BAD-TRAFFIC TCP port 0 traffic [**]
[Classification: Misc activity] [Priority: 3]
12/20-14:24:39.611549 2001:06c0:1973:0000:0000:0000:0000:0010:0 -> 2001:06c0:197
3:0000:0000:0000:0000:0001:20
TCP TTL:64 TOS:0x0 ID:0 IpLen:40 DgmLen:60
***A*R** Seq: 0x0  Ack: 0x1  Win: 0x0  TcpLen: 20

[**] [122:1:1] PSNG_TCP_PORTSCAN [**]
[Classification: Attempted Information Leak] [Priority: 2]
12/20-14:24:39.611555 2001:06c0:1973:0000:0000:0000:0000:0001 -> 2001:06c0:1973:
0000:0000:0000:0000:0010
PROTO:255 TTL:64 TOS:0x0 ID:0 IpLen:40 DgmLen:233

[**] [116:431:1] (snort_decoder) WARNING: ICMP6 type not decoded [**]
[Classification: Misc activity] [Priority: 3]
12/20-14:24:39.644985 2001:06c0:1973:0000:0000:0000:0000:0001 -> ff02:0000:0000:
0000:0000:0001:ff00:0010
IPV6-ICMP TTL:255 TOS:0x0 ID:0 IpLen:40 DgmLen:72
```

Figure 4.22: SNORT Transitional results for *portscan6.py*

NMAP6

The detection alert shown in Figure 4.23 and 4.10 illustrates SNORT's ability to detect NMAP6 TCP scans in both Transitional and "Native" environments. Figure 4.23 displays the Transitional alert results for **NMAP6**.

```
[**] [116:431:1] (snort_decoder) WARNING: ICMP6 type not decoded [**]
[Classification: Misc activity] [Priority: 3]
12/21-10:15:37.143184 2001:06c0:1973:0000:0000:0000:0000:0010 ->
2001:06c0:1973:0000:0000:0000:0000:0001
IPV6-ICMP TTL:255 TOS:0x0 ID:0 IpLen:40 DgmLen:72

[**] [122:1:1] PSNG_TCP_PORTSCAN [**]
[Classification: Attempted Information Leak] [Priority: 2]
12/21-10:15:37.143700 2001:06c0:1973:0000:0000:0000:0000:0001 ->
2001:06c0:1973:0000:0000:0000:0000:0010
PROTO:255 TTL:64 TOS:0x0 ID:0 IpLen:40 DgmLen:236
```

Figure 4.23: SNORT Transitional results for **NMAP6**

Toobig6

In the Transitional environment SNORT provided the same alerts as those shown in §4.1.2.

```
[**] [116:285:1] (snort_decoder) WARNING: ICMPv6 packet of type 2 (message
too big) with MTU field < 1280 [**]
[Classification: Generic Protocol Command Decode] [Priority: 3]
12/29-12:27:46.456683 2001:06c0:1973:0000:0000:0000:0000:0001 ->
2001:06c0:1973:0000:0000:0000:0000:0010
IPV6-ICMP TTL:255 TOS:0x0 ID:0 IpLen:40 DgmLen:1249

[**] [1:1394:12] SHELLCODE x86 inc ecx NOOP [**]
[Classification: Executable Code was Detected] [Priority: 1]
12/29-12:27:46.456683 2001:06c0:1973:0000:0000:0000:0000:0001 ->
2001:06c0:1973:0000:0000:0000:0000:0010
IPV6-ICMP TTL:255 TOS:0x0 ID:0 IpLen:40 DgmLen:1249

[**] [1:460:8] ICMP unassigned type 2 [**]
[Classification: Misc activity] [Priority: 3]
12/29-12:27:46.456683 2001:06c0:1973:0000:0000:0000:0000:0001 ->
2001:06c0:1973:0000:0000:0000:0000:0010
IPV6-ICMP TTL:255 TOS:0x0 ID:0 IpLen:40 DgmLen:1249
```

Figure 4.24: SNORT Transitional results for **Toobig6**

Flood_Advertise6

Figure 4.25 displays the detection results for **Flood_Advertise6**, where the three alerts listed appear to be from different IP address going to the all_host multicast address. Again, with closer inspection, however, each of these three addresses share the same first 80 bits (`fe80:0000:0000:0000:0218`) which could be an indication of one host generating random addresses to flood advertisements on the network segment.

```
[**] [116:431:1] (snort_decoder) WARNING: ICMP6 type not decoded [**]
[Classification: Misc activity] [Priority: 3]
12/21-11:28:14.579143 fe80:0000:0000:0000:0218:48ff:fe21:2785 ->
ff02:0000:0000:0000:0000:0000:0000:0001
IPV6-ICMP TTL:255 TOS:0x0 ID:0 IpLen:40 DgmLen:72

[**] [116:431:1] (snort_decoder) WARNING: ICMP6 type not decoded [**]
[Classification: Misc activity] [Priority: 3]
12/21-11:28:14.581503 fe80:0000:0000:0000:0218:02ff:fe4d:7ab4 ->
ff02:0000:0000:0000:0000:0000:0000:0001
IPV6-ICMP TTL:255 TOS:0x0 ID:0 IpLen:40 DgmLen:72

[**] [116:431:1] (snort_decoder) WARNING: ICMP6 type not decoded [**]
[Classification: Misc activity] [Priority: 3]
12/21-11:28:14.582091 fe80:0000:0000:0000:0218:99ff:fedc:7037 ->
ff02:0000:0000:0000:0000:0000:0000:0001
IPV6-ICMP TTL:255 TOS:0x0 ID:0 IpLen:40 DgmLen:72
```

Figure 4.25: SNORT Transitional results for **Flood_Advertise6**

Denial6

SNORT's detection results in the Transitional environment for **Denial6** test case 1 (large hop-by-hop headers with router-alert filled with unknown options), shown in Figure 4.26 are more accurate than those seen in Figure 4.15 from §4.1.2. In this case SNORT alerted on an unidentified IPv6 header options type.

```
[**] [116:279:1] (snort_decoder) WARNING: IPv6 header includes an undefined opti
on type [**]
[Classification: Generic Protocol Command Decode] [Priority: 3]
12/21-11:48:49.425340 2001:06c0:1973:0000:0000:0000:0000:0001 -> 2001:06c0:1973:
0000:0000:0000:0000:0010
IPV6-ICMP TTL:255 TOS:0x0 ID:516 IpLen:40 DgmLen:1480

[**] [1:18474:1] ICMPv6 Echo Request [**]
[Classification: Misc activity] [Priority: 3]
12/21-11:48:49.425340 2001:06c0:1973:0000:0000:0000:0000:0001 -> 2001:06c0:1973:
0000:0000:0000:0000:0010
IPV6-ICMP TTL:255 TOS:0x0 ID:516 IpLen:40 DgmLen:1480

[**] [116:279:1] (snort_decoder) WARNING: IPv6 header includes an undefined opti
on type [**]
[Classification: Generic Protocol Command Decode] [Priority: 3]
12/21-11:48:49.425769 2001:06c0:1973:0000:0000:0000:0000:0001 -> 2001:06c0:1973:
0000:0000:0000:0000:0010
IPV6-ICMP TTL:255 TOS:0x0 ID:516 IpLen:40 DgmLen:1480
```

Figure 4.26: SNORT Transitional results for **Denial6**

Fake_router6 With Fragmentation

Figure 4.27 displays the detection results from **Fake_router6** using fragmentation. What is seen with this alert is much like that from the alert seen with **Fragmentation6** in §4.1.2, with the exception that SNORT alerts on overlapping fragments in this attack.

```
[**] [123:10:1] (spp_frag3) Bogus fragmentation packet. Possible BSD attack [**]
[Classification: Attempted Administrator Privilege Gain] [Priority: 1]
12/21-11:21:41.691505 fe80:0000:0000:0000:0a00:27ff:feb3:9fb3 -> ff02:0000:0000:
0000:0000:0000:0000:0001
IPV6-FRAG TTL:255 TOS:0xE0 ID:17739787 IpLen:40 DgmLen:66
Frag Offset: 0x0000   Frag Size: 0x001A

[**] [123:8:1] (spp_frag3) Fragmentation overlap [**]
[Classification: Generic Protocol Command Decode] [Priority: 3]
12/21-11:21:41.691505 fe80:0000:0000:0000:0a00:27ff:feb3:9fb3 -> ff02:0000:0000:
0000:0000:0000:0000:0001
IPV6-FRAG TTL:255 TOS:0xE0 ID:17739787 IpLen:40 DgmLen:66
Frag Offset: 0x0000   Frag Size: 0x001A

[**] [123:10:1] (spp_frag3) Bogus fragmentation packet. Possible BSD attack [**]
[Classification: Attempted Administrator Privilege Gain] [Priority: 1]
12/21-11:21:46.688302 fe80:0000:0000:0000:0a00:27ff:feb3:9fb3 -> ff02:0000:0000:
0000:0000:0000:0000:0001
```

Figure 4.27: SNORT Transitional results for **Fake_router6** using fragmentation

Other Transitional Detection Results

Just as seen in Table 4.1 for the "Native" environment, in the Transitional environment SNORT presented "ICMP type not decoded" as an alert in *Alert.log* for each listed attack, which is an indication that the IPv6 packets from the attacks are getting processed correctly up to SNORT's Detection Engine, and thus a sign of detection.

Table 4.3 displays the results of all tests in the Transition test case.

Transitional Detection Matrix

			Detection		
Attack Type			BRO	BRO2	SNORT
	IPv4 Attack	**IPv6 Attack**			
Reconnaissance					
	Port Scan	Alive6(resolve)	No	No	Yes
	Port Scan	Alive6(Inv. Hdr)	No	No	Yes
	Port Scan	Alive6(Inv. Hop by Hop)	No	No	Yes
	portscan4.py	portscan6.py	Yes**	No+	Yes*
	NMAP	NMAP6(scan ports)	Yes**	No+	Yes
	NMAP	NMAP6(TCP scan)	Yes**	No	Yes
Unauthorized Access					
	ARP Poisoning(ICMP)	Fake_Advertise6	No	No	Yes
	DNS	Fake_DNS6	Yes%	Yes(w)	Yes
	Route Implanting	Toobig6	No	No	Yes
Host Configuration					
	DHCP DoS	DoS-new-IPv6	No	No	Yes
		Detect-new-IPv6	No	No	Yes
		Fake_DHCPS6	No	No	Yes
		NDPexhaust	No	No+	Yes
Broadcast Amplification Attacks(SMURF)					
	SMURF	SMURF6	No	No+	Yes
		RSMURF6	No	No+	Yes
Routing Attacks					
	Various DoS	Fake_Router6	No	No	No
		Fake_MLDrouter6(Sol.)	No	No	Yes
		Fake_MLDrouter6(Term.)	No	No	Yes
		Fake_MLD26(Query)	No	No	Yes
		Fake_MLD26(Add)	No	No	Yes
		Fake_MLD26(Delete)	No	No	Yes
		Kill_router6	No	No	No

Transitional Detection Matrix Cont.

			Detection		
Attack Type			BRO	BRO2	SNORT
	IPv4 Attack	**IPv6 Attack**			
Man in The Middle (MITM)					
	Various	Parasite6	No	No	Yes
Flooding					
	Various DoS	Denial6(lrg Hop by Hop)	No	No+	Yes
		Denial6(large dst Hdr)	No	No	Yes
	DHCP flooding	Flood_DHCPc6	No+	No+	Yes
	Various	Flood_Advertise6	Yes(w)	Yes(w)	Yes
		Flood_MLD6	No	No	Yes
		Flood_MLDrouter6	No	No	Yes
		Flood_Solicitate6(Network)	No	No	Yes
		Flood_Solicitate6(Target IP)	No	No	Yes
Rogue Devices					
	Rogue Router	Fake_Router6	No	No	No
		Fake_MLD6	No	No	Yes
		Fake_MLDrouter6(Adv.)	No	No	Yes
Fragmentation					
	Fragmentation	Fragmentation6	No	No	Yes
		Fake_Router6(Frag)	No	No	Yes
		Kill_Router6(Frag)	No	No	Yes
Exploit					
	Malware Transfer	FTP Malware	No	No	Yes#
	Various	Exploit6	No	No+	Yes
		Implementation6	No	Yes(w)	Yes

Table 4.3: Transitional Detection Matrix —— Notes:* our Scapy portscan6 scanned TCP port 0 which was detected by SNORT, ** BRO detected after we enabled event tracing, # Read string "Malware.exe", + dropped packet activity in *Notice.log*, (w) *Weird.log* activity, % showed as portscan.

4.3 BRO/SNORT and FTP

During the FTP transfers with the Slackbot malware (discussed in §3.3.1), neither BRO nor SNORT detected the Slackbot malaware.exe payload with their deep packet inspection capabilities. The only detection that occurred during any of the transfers was SNORT alerting that an executable was being transferred when the file was named *malware.exe*. No alert was given when the extension (.exe) was removed or when the file was zipped, this indicates SNORT was simply alerting on a filename with an executable (.exe) extension being transferred over FTP. We ran the same transfers on an IPv4 network setup with the same results.

4.3.1 BRO FTP Results

The preceding result is not particularly useful since more varieties of malware samples would be needed to be sent to test payload inspection detection capabilities in BRO and SNORT. However, we deemed that since the results were mostly the same moving ahead with more of these tests did not seem useful. However, when sending the FTP transfers over an IPv6 network, BRO returned some error messages that suggested it was having difficulty processing these events in IPv6, just as in the indications seen in §4.1 and §4.2. Figure 4.28 shows the actual error messages received.

```
tester@BRO-VirtualBox:~/logs$ sudo /usr/local/bro/bin/bro -i eth0 /usr/local/bro/share/bro/site/local.bro -t trace-1-1282011-v6.log
Execution tracing ON.
listening on eth0
1323371431.633088 error: bad dotted address: 2001:6c0:1973::1|42302|
1323371905.070753 error: bad dotted address: 2001:6c0:1973::1|34388|
1323371920.363719 error: bad dotted address: 2001:6c0:1973::1|55261|
1323371925.002372 error: bad dotted address: 2001:6c0:1973::1|40376|
1323371934.444545 error: bad dotted address: 2001:6c0:1973::1|41219|
1323371940.094269 error: bad dotted address: 2001:6c0:1973::1|42052|
```

Figure 4.28: BRO "Bad dotted address" error message

The BRO event logs contained very few of the packets during the transfer which again led us to believe that BRO had more fundamental problems with IPv6, at least where FTP transfers were concerned. BRO was now also printing some error messages captured in Figure 4.28. We ran the *malware.exe* transfer again, but this time, while debugging BRO using GDB [57] to get a better idea of what was happening. One of the breakpoints we set was in a file called *net_util.cc* on line 238. Figure 4.29 has the contents of *dotted_to_addr* in *net_util.cc* which takes a const char* as a parameter and returns a unit_32 value. The function converts an IPv4 string into a unsigned integer representation of an IPv4 address. However, in this case it was being passed an IPv6 string and returning 0 after failing the check in the second if statement. Returning 0 here resulted in failing to create the event properly in BRO's event log. Another problem with this

type of traffic is that the IP address is contained within the the data portion of the FTP control packets, which is the same problem that is seen in Network Address Translation (NAT), thus obscuring the addresses to the NIDS.

```
224 uint32 dotted_to_addr(const char* addr_text)
225     {
226     int addr[4];
227
228     if ( sscanf(addr_text,
229                 "%d.%d.%d.%d", addr+0, addr+1, addr+2, addr+3) != 4 )
230             {
231             error("bad dotted address:", addr_text );
232             return 0;
233             }
234
235     if ( addr[0] < 0 || addr[1] < 0 || addr[2] < 0 || addr[3] < 0 ||
236          addr[0] > 255 || addr[1] > 255 || addr[2] > 255 || addr[3] > 255 )
237             {
238             error("bad dotted address:", addr_text);
239             return 0;
240             }
241
242     uint32 a = (addr[0] << 24) | (addr[1] << 16) | (addr[2] << 8) | addr[3];
243
244     // ### perhaps do gethostbyaddr here?
245
246     return uint32(htonl(a));
247     }
```

Figure 4.29: BRO *Dotted_to_Addr* function in *Net_util.cc*

net_util.cc does have a function called *dotted_to_addr6*, but it was not being called in this case. BRO also had a very difficult time processing IPv6 packets with FTP. It reported that only three packets were processed after the BRO process was terminated at the end of this FTP test.

4.4 SNORT Rules

The detection matrix displayed in Table 4.2 shows BRO at a 10 percent detection success rate and SNORT at a 52 percent detection success rate. However, these numbers only paint part of the picture. In the previous two sections we outlined the difficulties BRO had even adding IPv6 traffic to its event logging. In the cases where SNORT did not detect, examination of its event logging indicated that these events were being captured effectively. The events actually looked nearly identical in the logs in most of the cases. In other examples SNORT logs showed detectable events, such as in Figure 4.24 or Figure 4.10. In both these cases only rule problems exist, where a rule can be written to alert on these events, meaning SNORT's underlying engine is still sound.

Taking into account this ability to develop rules for detection, the revised detection rate for SNORT sits at 95 percent for "Native" and 93 percent in the Transitional environment. In this sense, SNORT is more IPv6 capable as rules just need to be written, but in contrast to BRO,

fundamental code changes and/or debugging should not be necessary. Figure 4.30 displays a few simple externally developed IPv6 related SNORT rules that are not part of the built in rule set, however the addition of these rules would improve SNORT's IPV6 readiness.

```
alert ip icmp any -> any any (msg:"IPv6 ICMP Router Advertisement"; itype:134;
                                    classtype:icmp-event; sid:2000001; rev:1;)

alert ip any any -> any any (msg:"TTL or Hop Limit = 50"; ttl:50;
                                classtype:attempted-recon; sid:2000002; rev:1;)
```

Figure 4.30: SNORT rules. From [23]

4.5 SNORT 2.9.0.5 Bug

While running **Implementation6** we discovered a warning message we had not seen before, shown in Figure 4.31. This message by itself is not alarming, however when we sent the next attack we discovered that SNORT was no longer processing packets. One of the test cases in **Implementation6** had allowed our next attack to circumvent the NIDS. In fact, the NIDS had failed open which means that it did not crash and sat in a state of infinite recursion. Failing in this state would be a serious problem in a live NIDS, allowing an attacker to quietly bypass SNORT and easily slip any subsequent attacks into the victim's network.

Since there are more than 40 tests in this attack we isolated the test responsible for the problem. To isolate, we ran each test individually while monitoring SNORT's reaction by running it in the GDB debugger. After running each test case individually, we discovered that test 3 "128 hop-by-hop headers" was responsible for the SNORT failure. This test case uses 128 hop-by-hop headers, each with a value of zero to ensure that the OS's IPv6 stack is processing the headers correctly.

Figure 4.31 shows the output of SNORT after entering recursion. Note that this is the only indication operators would see, and may possibly miss if not using the verbose logging option.

```
WARNING: decoder got too many layers; next proto is 7
WARNING: decoder got too many layers; next proto is 7
WARNING: decoder got too many layers; next proto is 7
WARNING: decoder got too many layers; next proto is 7
WARNING: decoder got too many layers; next proto is 7
WARNING: decoder got too many layers; next proto is 7
WARNING: decoder got too many layers; next proto is 7
WARNING: decoder got too many layers; next proto is 7
WARNING: decoder got too many layers; next proto is 7
WARNING: decoder got too many layers; next proto is 7
```

Figure 4.31: SNORT output for *Implementation6* with 128 hop-by-hop headers

The Problem

After searching through the `SNORT-2.9.0.5/src/` directory in order to trace the "decoder got too many layers; next proto is 7" warning message, we found *decode.c* to be the source of the problem. We discovered that SNORT was being sent into infinite recursion between `DecodeIPV6Extensions` and `DecodeIPV6Options`. The normal process for decoding extension headers is listed in Figure 4.32.

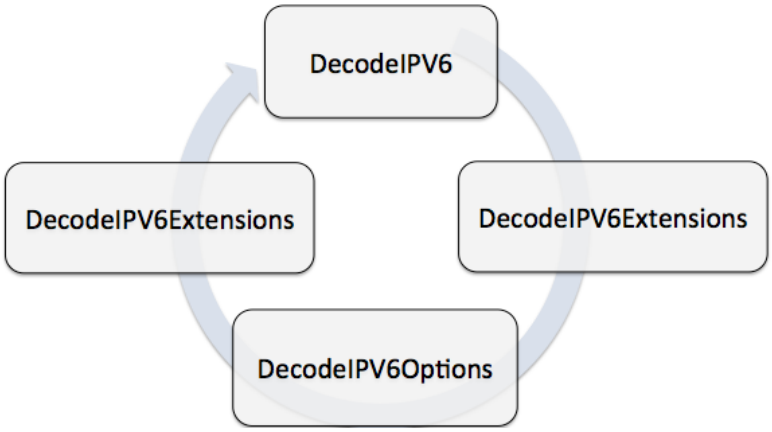

Figure 4.32: Normal processing of extension headers in *decode.c*

Each extension header is processed by `DecodeIP6`, sent to `DecodeIPV6Extensions`, then if it has any options it is passed to `DecodeIPV6Options` where it is finally passed back to `DecodeIPV6Extensions`. It is in this process where we found the problem.

As we traced the operation of *decode.c* during test 3 we discovered that at a given point (number of hop-by-hop headers) it was recursing between two functions `DecodeIPV6Extensions` and `DecodeIPV6Options`. With further tracing and stepping through break points we finally discovered the problem function was `DecodeIPV6Options`.

Figure 4.33 shows the first 43 lines of the `DecodeIPV6Options` function in *Decode.c*. At line 5 *hdrlen* is set to 0, while line 15 shows the recursion entry point.

```
1  void DecodeIPV6Options(int type, const uint8_t *pkt, uint32_t len, Packet *p)
2  {
3      printf("Entering: %s (len: %d)\n", __func__, len);
4      IP6Extension *exthdr;
5      uint32_t hdrlen = 0;
6
7      /* This should only be called by DecodeIPV6 or DecodeIPV6Extensions
8       * so no validation performed.  Otherwise, uncomment the following: */
9      /* if(IPH_IS_VALID(p)) return */
10
11     pc.ipv6opts++;
12
13     /* Need at least two bytes, one for next header, one for len. */
14     /* But size is an integer multiple of 8 octets, so 8 is min.  */
15     if(len < sizeof(IP6Extension))
16     {
17     printf("Len %d is less than sizeof(IP6Extension: %d)\n",
18         len, sizeof(IP6Extension));
19         DecoderEvent(p, DECODE_IPV6_TRUNCATED_EXT,
20                      DECODE_IPV6_TRUNCATED_EXT_STR,
21                      1, 1);
22         return;
23     }
24
25     exthdr = (IP6Extension *)pkt;
26
27     printf("Determine IP6 extension.  Count: %d  Max: %d\n",
28     p->ip6_extension_count, IP6_EXTMAX);
29     if(p->ip6_extension_count < IP6_EXTMAX)
30     {
31         p->ip6_extensions[p->ip6_extension_count].type = type;
32         p->ip6_extensions[p->ip6_extension_count].data = pkt;
33
34         // TBD add layers for other ip6 ext headers
35         switch (type)
36         {
37             case IPPROTO_HOPOPTS:
38                 if (len < sizeof(IP6HopByHop))
39                 {
40                     DecoderEvent(p, DECODE_IPV6_TRUNCATED_EXT,
41                                  DECODE_IPV6_TRUNCATED_EXT_STR,
42                                  1, 1);
43                     return;
```

Figure 4.33: First 43 lines of code for `DecodeIPV6Options`

When `DecodeIPV6Options` is called, `hdrlen` is set to zero and then based on a set of switch cases, is then set appropriately to be passed back to `DecodeIPV6Extensions`.
As `DecodeIPV6Options` is processing hop-by-hop headers a count of the extension headers (`p->ip_extension_count`) is compared to a set value for the maximum number of extension headers (`IP6_EXTMAX`). This would not normally be an problem, however since `IP6_EXTMAX` is hardcoded to 40, when the `p->ip_extension_count` exceeds this set maximum, the switch cases that set `hdrlen` are never reached. In this event, `hdrlen` is never updated.

Figure 4.34 shows a few of the switch cases where `hdrlen` is set and passed back to `DecodeIPV6Extensions`, where the next header processed. At this point, `DecodeIPV6Extensions` calls back to `DecodeIPV6Options` until `IP6_EXTMAX` is hit and then it loops in this state indefinitely between `DecodeIPV6Options` and `DecodeIPv6Extensions`.

```
29      if(p->ip6_extension_count < IP6_EXTMAX)
30      {
31          p->ip6_extensions[p->ip6_extension_count].type = type;
32          p->ip6_extensions[p->ip6_extension_count].data = pkt;
33
34          // TBD add layers for other ip6 ext headers
35          switch (type)
36          {
37              case IPPROTO_HOPOPTS:
38                  if (len < sizeof(IP6HopByHop))
39                  {
40                      DecoderEvent(p, DECODE_IPV6_TRUNCATED_EXT,
41                                   DECODE_IPV6_TRUNCATED_EXT_STR,
42                                   1, 1);
43                      return;
44                  }
45                  hdrlen = sizeof(IP6Extension) + (exthdr->ip6e_len << 3);
46                  if (CheckIPV6HopOptions(pkt, len, p) == 0)
47                      PushLayer(PROTO_IP6_HOP_OPTS, p, pkt, hdrlen);
48                  break;
49
50              case IPPROTO_DSTOPTS:
51                  if (len < sizeof(IP6Dest))
52                  {
53                      DecoderEvent(p, DECODE_IPV6_TRUNCATED_EXT,
54                                   DECODE_IPV6_TRUNCATED_EXT_STR,
55                                   1, 1);
56                      return;
57                  }
58                  if (exthdr->ip6e_nxt == IPPROTO_ROUTING)
59                  {
60                      DecoderEvent(p, DECODE_IPV6_DSTOPTS_WITH_ROUTING,
61                                   DECODE_IPV6_DSTOPTS_WITH_ROUTING_STR,
62                                   1, 1);
63                  }
64                  hdrlen = sizeof(IP6Extension) + (exthdr->ip6e_len << 3);
65                  if (CheckIPV6HopOptions(pkt, len, p) == 0)
66                      PushLayer(PROTO_IP6_DST_OPTS, p, pkt, hdrlen);
67                  break;
68
69              case IPPROTO_ROUTING:
```

Figure 4.34: Switch Cases in `DecodeIPV6Options`

Figure 4.35 illustrates the recursion between `DecodeIPV6Options` and `DecodeIPV6Extensions`, where `hdrlen` is always zero.

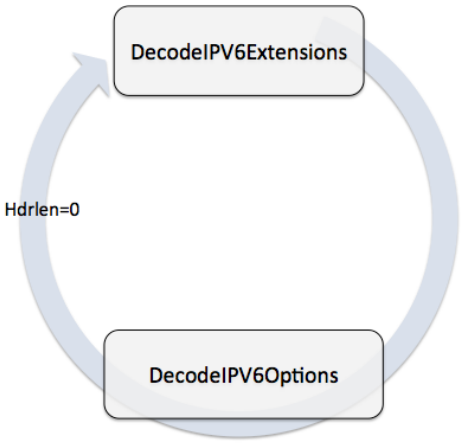

Figure 4.35: Illustration of recursion in `DecodeIPV6Options`

Possible Fix

Figure 4.36 displays a potential simple fix for *decode.c* that prevents `DecodeIPV6Options` from looping indefinitely once `ip6_extension_count` is equal to or greater than `IP6_EXTMAX` by adding an `else` statement to the function that allows it to return when `hdrlen` has not be set. Note that this is not the only solution, only a quick simple fix. In fact, after researching this problem further, we discovered that the Sourefire team had fixed this in the current development version (2.9.2.1) released 24 January, 2012.

```
128
129             case IPPROTO_AH:
130                 /* Auth Headers work in both IPv4 & IPv6, and their lengths are
131                    given in 4-octet increments instead of 8-octet increments. */
132                 hdrlen = sizeof(IP6Extension) + (exthdr->ip6e_len << 2);
133                 break;
134
135             default:
136                 hdrlen = sizeof(IP6Extension) + (exthdr->ip6e_len << 3);
137                 break;
138         }
139
140         p->ip6_extension_count++;
141     }
142     /* Note: potential 128 hop-by-hop headers "fix" */
143     else {
144     printf("Should return because I never set hdrlen\n");
145     return;
146          }
147
148     if(hdrlen > len)
149     {
150         DecoderEvent(p, DECODE_IPV6_TRUNCATED_EXT,
151                      DECODE_IPV6_TRUNCATED_EXT_STR,
152                      1, 1);
153         return;
154     }
155
156     DecodeIPV6Extensions(*pkt, pkt + hdrlen, len - hdrlen, p);
157 }
158
```

Figure 4.36: Example code for possible fix for SNORT 128 invalid hop-by-hop header bug

Figure 4.37 displays the SoureFire team's fix for *decode.c* in version 2.9.2.1.

```
void DecodeIPV6Options(int type, const uint8_t *pkt, uint32_t len, Packet *p)
{
    IP6Extension *exthdr;
    uint32_t hdrlen = 0;

    /* This should only be called by DecodeIPV6 or DecodeIPV6Extensions
     * so no validation performed.  Otherwise, uncomment the following: */
    /* if(IPH_IS_VALID(p)) return */

    pc.ipv6opts++;

    /* Need at least two bytes, one for next header, one for len. */
    /* But size is an integer multiple of 8 octets, so 8 is min.  */
    if(len < sizeof(IP6Extension))
    {
        DecoderEvent(p, DECODE_IPV6_TRUNCATED_EXT,
                     DECODE_IPV6_TRUNCATED_EXT_STR,
                     1, 1);
        return;
    }

    if ( p->ip6_extension_count >= IP6_EXTMAX )
    {
        DecoderEvent(p, DECODE_IP6_EXCESS_EXT_HDR,
                     DECODE_IP6_EXCESS_EXT_HDR_STR,
                     1, 1);
        return;
    }
```

Figure 4.37: SNORT 2.9.2.1 fix to 128 hop-by-hop header bug. From [23]

4.6 Fuzz Testing

Fuzz testing was completed as discussed in §3.5. During this testing 19 test cases were conducted and over 39 million packets were sent. These fuzz tests covered ICMPv6 services (echo request, neighbor solicitations and advertisements, and router advertisements), MLD services (report, done, and query packets), flag and byte type values, fragmentation, hop-by-hop headers, as well as router alerts and jumbo packet headers.

As discussed in §3.5, each NIDSs was run inside GDB during fuzzing while we waited for faults created by the fuzzed packets. During this process, we encountered no ill effects to fuzzing from any of our selected NIDS. Since our intent was to fuzz test the NIDSs reactions to changes in the traffic they were monitoring, any impact on the target host was not recorded. However during testing, we observed no impact on the target host either. Unfortunately, upon completion, the results of our fuzzing revealed no impact or data. Although these results were disappointing, they did aid us in answering the question of IPv6 readiness for each of our NIDS.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 5:
Conclusions

In this thesis we set out to answer two primary questions: which exploits that exist with IPv4 signatures are feasible or infeasible with conversion to IPv6, and whether popular open source NIDSs detect these IPv6 attacks. Both of these questions derive from a single requirement: are commonly used Network Intrusion Detection Systems ready for IPv6? To answer this question we examined common IPv4 attacks and exploits and classified them in thirteen attack categories. We then converted IPv4 attacks from each class their IPv6 equivalent. When direct conversion was not possible, we used attacks analogous to their IPv4 counterpart. Our IPv6 attacks consisted of a combination of SCAPY [31] scripts and programs from the THC IPv6-attack-toolkit [32].

To evaluate the converted attacks, we created a controlled virtual test bed and configured two two open source NIDSs (BRO and SNORT). Each attack was sent in the testing environment, followed by an exhaustive process of NIDS and systems log inspection in order to provide detailed detection results and to arrive at final conclusions.

It is important to note that our research was aimed both at a "native" IPv6 environment as well as a "transitional" one. By transitional, we are referring to the period of time that IPv4 and IPv6 will coexist by means of any transition mechanisms that may be utilized. This transitional period, where the use of Tunnels, Translation, and Transition mechanisms will be required, is intended to bridge the gap between where we are today and when all systems are IPv6 or "native." It is important to understand the distinction between the native and transitional periods, as well as the vulnerabilities associated with each.

Our original intent had each NIDS being configured as an "out of the box" solution in order to give us an initial picture of readiness. However, in reality there are quite a few configuration changes that must be made to each NIDS. One such configuration change required had to do with the reconnaissance attack or port scan. In order for SNORT to detect these scans, operators need to enable, via configuration, **sfPortscan** in *snort.conf*. This was just one of many issues that led us to conclude that neither of these NIDSs are "out of the box" solutions for IPv6.

For example, rule/policy "tuning" will always need to take place. Thus, our definition of readiness is one where the NIDS is configured to detect our IPv4 attacks (for ground-truth) and is configured for IPv6.

Overall IPv6 readiness was determined from actual attack detection, or the ability to detect with slight modification (e.g., rules, signatures, or policies). From our research results shown in Table 4.2 we determined that SNORT initially detected 52 percent of attacks sent. We also determined that the non-alerted attacks, the remaining 48 percent, could be easily detected with locally written rules. Any detection issues discovered with SNORT were in its configuration or rules, which can be instituted by system administrators, and is therefore not a fundamental software or design limitation. This leads to the determination that SNORT is indeed IPv6 ready, requiring only a handful of rules and configuration changes to post a detection success rate of 95 percent in a "Native" environment.

BRO on the other hand had an 8 percent detection success rate. This detection success rate would have been lower had we not enabled built in "event tracing" which seemed to enable port scan detection. In an attempt to be thorough, an exhaustive debugging process was used to determine why BRO performed poorly. We found that in many cases BRO is not rebuilding IPv6 events in the EVENT ENGINE, and that in other cases BRO is not appropriately dealing with IPv6 addresses themselves. For example, when we transferred Malware.exe via FTP, the function *Dotted_to_Address* in *Net_Util.cc* returns a "Bad Dotted Address" error. Another suspicious indication from BRO was the actual packets received versus the reported packets received by BRO. When sending *portscan6.py*, we transmitted over 6000 packets, however BRO reported on only 46 of those packets, despite logging all of the packets in the *event.bst* log. We were unable to determine the root cause of this issue, however we believe it to be related to IPv6 stream reconstruction. This event reconstruction problem is not an implementation issue, it is more fundamental in nature and will require more development from the BRO design team. When considering the IPv6 readiness of BRO, the issues we discovered along with our noted detection percentage lead to the conclusion that BRO is not yet IPv6 ready.

Figure 5.1 summarizes our results and shows the detection percentage for attacks in both the "Native" and Transitional environments.

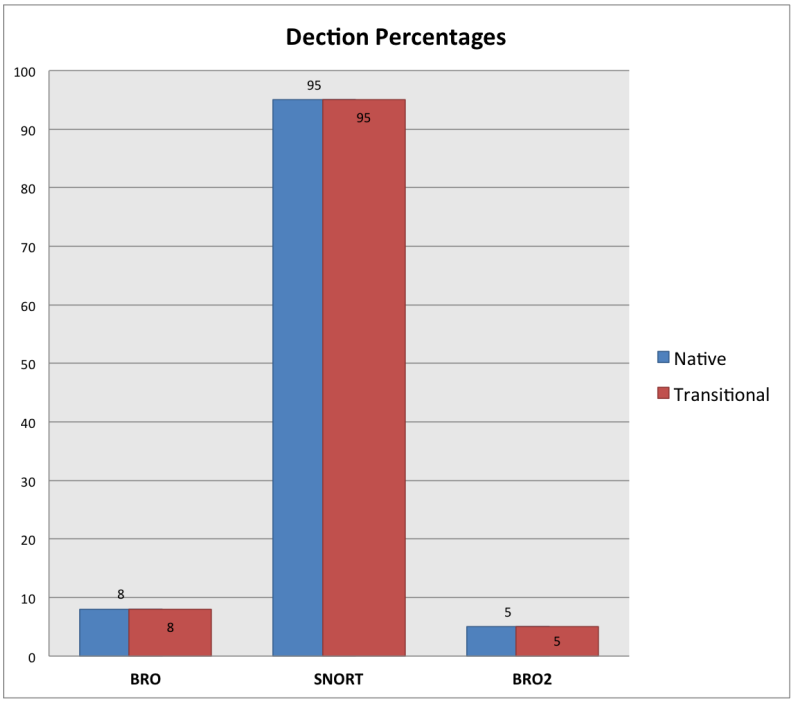

Figure 5.1: IPv6 attack detection percentages

5.0.1 A Bug in SNORT

The discovery of a bug in SNORT 2.9.0.5 during testing brings to light a few key points. First, it is possible cause this version of SNORT by stacking together enough, in this case more than 40, hop-by-hop extension headers. After failing open, SNORT is no longer operational, allowing an attacker to fully circumvent the NIDS. The result of sending these hop-by-hop headers, as discussed in §4.5, is the NIDS entering a state of infinite recursion and not being able to process any subsequent packets. To further complicate matters, while in this state of recursion SNORT appears to be running correctly, giving the operator no indication that it is not actually processing traffic. The end result allows a hacker to evade the NIDS and pass any traffic that they want onto a victim's network completely undetected. This becomes a serious security problem for anyone running this or an earlier version of the SNORT NIDS. The silver lining here is that the current version of SNORT (2.9.2.1 as of the time of this writing) released on 24 January, 2012, inadvertently fixed this problem while attempting to fix an unrelated problem with Teredo. The second key point is that even though SNORT is IPv6 ready there may still be similar vulnerabilities in the code or new ways in which to evade it. This requires constant update management by both SNORT developers and users. Finally, this bug highlights the importance of research and testing (e.g., like that in this thesis and Fuzz testing).

5.0.2 Fuzzing

We had expected that in response to Fuzz tests conducted, one of the NIDSs would have alarmed, halted, been circumvented, or given some other than normal indication, however this was not the case. Even though the Fuzz testing results seen in §4.6 were not what we expected, there is value in Fuzzing. The Fuzz tests performed in this thesis aided in the development of an IPv6 readiness conclusion and possibly set the stage for further Fuzzing. Our Fuzz tests were limited to services (e.g., ICMPv6, hop-by-hop headers, flags) and did not go any more granular. It is possible to generate Fuzzed packets that are more granular in order to test the NIDS's reaction to all kinds of altered and invalid traffic. We have left this type of Fuzz testing for future work.

5.1 Recommendations

This section lists recommendations based on our research data and conclusions in order to improve IPv6 readiness and detection performance for the selected NIDSs.

- Neither NIDSs implement IPv6 by default. If the installer does not carefully research configuration options before installation, they will not get IPv6 support. This could be alleviated by simply enabling IPv6 support by default, or possibly making it easily selectable for those users and administrators who may have specific protocol needs. As support for IPv6 grows, so will the need for the NIDSs that protect it.

- Development of IPv6 specific rules is needed. A drafted set of default IPv6 rules would bolster the NIDS and enhance out of the box IPv6 readiness. This set of rules could be implemented, stored, and developed for each NIDS, or official as part of cross-NIDS, standardized, rule sets such as SNORT's VRT [23].

- Continuously monitor vulnerability or exploit development sites. For example, the CVE [53] or THC [32] could provide powerful reference tools for NIDS developers in order to stay on top of attack and vulnerability trends.

- Existing tool kits and exploits, such as those found on the THC or Backtrack [59], could be used as a NIDS and rule set regression tool. Tools found on these sites should be tested and implemented during the version development process. This practice would greatly improve NIDS IPv6 readiness.

- While investigating the bug in SNORT 2.9.0.5, we discovered that the NIDS failed open allowing all subsequent traffic after the fault to pass without report. In cases like this, in order to improve security and protection, we recommend that NIDSs fail closed. In other words, where possible we suggest that system faults shut down the NIDS in such a way that the operator/administrator can not help but notice. This process improvement will not only improve the protection offered by the NIDS but also enhance bug reporting and development.

5.2 Future Work

In our research we have done the ground work and have made an initial determination of the IPv6 readiness of the tested NIDSs. Future work to enhance or complete our effort along with actually making the NIDSs IPv6 ready and the discovery of additional attacks may be accomplished in the following ways:

- Testing should continue in the area of exploits. This should include payload delivery, remote execution, and transitional mechanisms such as tunnels and translation devices. Increase effort should be made to monitor and use developments listed in the CVE's [53]

- BRO development to increase IPv6 support in the areas of the items discovered in this research as well as those that will be discovered with more exhaustive debugging. For both version 1.5.3 and version 2.0, more detailed work on IPv6 will be needed and it is rumored that the next iteration of BRO development will be solely to work these IPv6 issues.

- SNORT rules/signatures and BRO polices should be written to enhance detection. The bolstering of built-in databases and repositories such as VRT [23] for specific IPv6 issues and attacks would enhance the protection offered by both NIDS tested. For SNORT, additional rules to detect Router Advertisements, and Neighbor Solicitations as well as specific IPv6 attacks would provide a good foundation for protection.

- Finding additional IPv6 attack vectors remains an area of great need. Fuzz [10] testing could be used as a tool to discover new attacks, as well as to determine the impact on each of the NIDSs, and may enhance development in the areas of Intrusion Detection and Prevention.

- Examination of False Alarm rates would greatly increase the readiness and performance of each NIDS. This should include a close look into the false-positive and false-negative rates of not only this research's test bed environment, but also real world IPv6 traffic environments.

Finally, the growing adoption of IPv6, along with the DoD and industry push for it, enhances the need for IPv6 readiness in our Networks and Intrusion Detection Systems. It is hoped that the research provided in this thesis lays the foundation for an increased awareness and work towards general IPv6 security posture readiness.

REFERENCES

[1] S. Deering and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification." RFC 2460 (Draft Standard), Dec. 1998. Updated by RFCs 5095, 5722, 5871.

[2] J. Postel, "Internet Protocol." RFC 791 (Standard), Sept. 1981. Updated by RFC 1349.

[3] Hurricane Electric Internet Services, "Hurricane electric ipv4 exhaustion counters." http://ipv6.he.net/statistics, April 2011.

[4] K. Claffy, "Tracking IPv6 Evolution: Data We Have and Data We Need," *ACM SIGCOMM Computer Communication Review (CCR)*, pp. 43–48, Jul 2011.

[5] J. St Sauver, "ipv6 technical challenges," NSFTA Canada, Montreal, Quebec, Nov. 2010.

[6] A. Choudhary, "In-depth analysis of ipv6 security posture," in *CollaborateCom*, pp. 1–7, 2009.

[7] D. Zagar and K. Grgic, "Ipv6 security threats and possible solutions," World Automation Congress, 2006.

[8] S. Hogg and E. Vyncke, *IPv6 Security*. Cisco Press, 1st ed., 2008.

[9] S. Kent and R. Atkinson, "Security Architecture for the Internet Protocol." RFC 2401 (Proposed Standard), Nov. 1998. Obsoleted by RFC 4301, updated by RFC 3168.

[10] B. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Commun. ACM*, vol. 33, pp. 32–44, December 1990.

[11] P. Hart, "A management perspective of the department of defense (dod) internet protocol version 6(ipv6) transition plan, where it is today, and where it needs to be by the year 2008," Master's thesis, Naval Postgraduate School, Monterey, CA, 2006.

[12] IAB and IESG, "IAB/IESG Recommendations on IPv6 Address Allocations to Sites." RFC 3177 (Informational), Sept. 2001. Obsoleted by RFC 6177.

[13] S. Chozos, "Implementation and analysis of a threat model for ipv6 host autoconfiguration," Master's thesis, Naval Postgraduate School, Monterey, CA, 2006.

[14] R. Hinden and S. Deering, "Internet Protocol Version 6 (IPv6) Addressing Architecture." RFC 3513 (Proposed Standard), Apr. 2003. Obsoleted by RFC 4291.

[15] IANA, "Iana-port numbers." http://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xml, Oct. 2011.

[16] J. Postel, "Internet Control Message Protocol." RFC 792 (Standard), Sept. 1981. Updated by RFCs 950, 4884.

[17] A. Conta, S. Deering, and M. Gupta, "Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification." RFC 4443 (Draft Standard), Mar. 2006. Updated by RFC 4884.

[18] T. Narten, E. Nordmark, W. Simpson, and H. Soliman, "Neighbor Discovery for IP version 6 (IPv6)." RFC 4861 (Draft Standard), Sept. 2007. Updated by RFC 5942.

[19] R. Hinden and S. Deering, "IP Version 6 Addressing Architecture." RFC 4291 (Draft Standard), Feb. 2006. Updated by RFCs 5952, 6052.

[20] S. Thomson and T. Narten, "IPv6 Stateless Address Autoconfiguration." RFC 2462 (Draft Standard), Dec. 1998. Obsoleted by RFC 4862.

[21] SourceForge, "Jpcap." http://sourceforge.net/projects/jpcap/, Sept. 2011.

[22] J. Kurose and K. Ross, *Computer Networking: A Top-Down Approach*. USA: Addison-Wesley Publishing Company, 5th ed., 2009.

[23] Sourcefire, "Snort homepage." http://www.snort.org, April 2011.

[24] The Bro Project, "Bro homepage." http://www.bro-ids.org, April 2011.

[25] A. Mitra, W. Najjar, and L. Bhuyan, "Compiling pcre to fpga for accelerating snort ids," in *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, ANCS '07, (New York, NY, USA), pp. 127–136, ACM, 2007.

[26] A. Arboleda and C. Bedon, "Snort diagrams for developers," *Tech. report, Universidad del Cauca - Colombia*, 2005.

[27] R. Sommer and V. Paxson, "Enhancing byte-level network intrusion detection signatures with context," *Proceedings of the 10th ACM conference on Computer and communication security CCS 03*, p. 262, 2003.

[28] M. C. Moya, "Analysis and evaluation of the snort and bro network intrusion detection systems," Sept. 2008.

[29] SourceForge, "Tcpdump." http://www.tcpdump.org, Feb. 2011.

[30] Internet Society, "World ipv6 day." http://www.worldipv6day.org, June 2011.

[31] SecDev, "Scapy homepage." http://www.secdev.org/projects/scapy, April 2011.

[32] V. Hauser, "Thc-ipv6 attack tool kit." http://www.thc.org/thc-ipv6, April 2011.

[33] C. Caicedo, J. Joshi, and S. Tuladhar, "Ipv6 security challenges," *Computer*, vol. 42, pp. 36 –42, feb. 2009.

[34] S. Convery and D. Miller, "Ipv6 and ipv4 threat comparison and best-practice evaluation (v1.0)," *www.seanconvery.comv6-v4-threats.pdf*, March 2004.

[35] M. Crawford and B. Haberman, "IPv6 Node Information Queries." RFC 4620 (Experimental), Aug. 2006.

[36] D. Morr, "Thoughts on ipv6 security, take two." http://www.personal.psu.edu/dvm105/blogs/ipv6/2009/05/thoughts-on-ipv6-security-take.html, May 2009.

[37] S. Bellovin, A. Keromytis, and B. Cheswick, "Worm propagation strategies in an ipv6 internet," *USENIX ;login*, vol. 31, pp. 70–76, February 2006.

[38] S. Staniford, V. Paxson, and N. Weaver, "How to own the internet in your spare time," in *Proceedings of the 11th USENIX Security Symposium*, (Berkeley, CA, USA), pp. 149–167, USENIX Association, 2002.

[39] E. Nerakis, "Ipv6 host fingerprint," Master's thesis, Naval Postgraduate School, Monterey, CA, 2006.

[40] R. Hinden and S. Deering, "IPv6 Multicast Address Assignments." RFC 2375 (Informational), July 1998.

[41] P. Ferguson and D. Senie, "Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing." RFC 2827 (Best Current Practice), May 2000. Updated by RFC 3704.

[42] J. Arkko, J. Kempf, B. Zill, and P. Nikander, "SEcure Neighbor Discovery (SEND)." RFC 3971 (Proposed Standard), Mar. 2005.

[43] M. Pohl, "Experimentation and evaluation of ipv6 secure neighbor discovery protocol," Master's thesis, Naval Postgraduate School, Monterey, CA, 2007.

[44] A. Conta and S. Deering, "Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification." RFC 2463 (Draft Standard), Dec. 1998. Obsoleted by RFC 4443.

[45] Hurricane Electric Internet Services, "Hurricane electric homepage." http://ipv6.he.net/, April 2011.

[46] C. Aoun and E. Davies, "Reasons to Move the Network Address Translator - Protocol Translator (NAT-PT) to Historic Status." RFC 4966 (Informational), July 2007.

[47] Oracle, "Virtualbox homepage." https://www.virtualbox.org/, Sept 2011.

[48] Microsoft Corporation, "Microsoft windows 7 homepage." http://windows.microsoft.com/en-us/windows7/products/home, Sept 2011.

[49] Canonical Ltd., "Ubuntu homepage." http://www.ubuntu.com/, Sept 2011.

[50] NMAP, "Nmap homepage." http://nmap.org/, Sept 2011.

[51] S. Deering, W. Fenner, and B. Haberman, "Multicast Listener Discovery (MLD) for IPv6." RFC 2710 (Proposed Standard), Oct. 1999. Updated by RFCs 3590, 3810.

[52] R. Vida and L. Costa, "Multicast Listener Discovery Version 2 (MLDv2) for IPv6." RFC 3810 (Proposed Standard), June 2004. Updated by RFC 4604.

[53] MITRE, "Cve homepage." http://cve.mitre.org/, Jan 2012.

[54] C. Evans, "Vsftpd setup page." https://security.appspot.com/vsftpd.html, Dec 2011.

[55] Symantec, “Symantec slackbot page.” http://www.symantec.com/security_response/writeup.jsp?docid=2001-100912-0421-99, Dec 2011.

[56] Jotti, “Jotti homepage.” http://virusscan.jotti.org, Dec 2011.

[57] Free Software Foundation Inc., “Gnu project debugger homepage.” http://www.gnu.org/s/gdb/, Dec 2011.

[58] Wireshark Foundation, “Wireshark homepage.” http://www.wireshark.org/, Dec 2011.

[59] BackTrack, “Backtrack-linux homepage.” http://www.backtrack-linux.org/, Jan 2012.

Referenced Authors

THIS PAGE INTENTIONALLY LEFT BLANK

Initial Distribution List

1. Dudly Knox Library
   Naval Postgraduate School
   Monterey, California

2. Defense Technical Information Center
   Ft. Belvoir, Virginia

3. Head, Information Operations and Space Integration Branch,
   PLI/PP&O/HQMC, Washington, DC